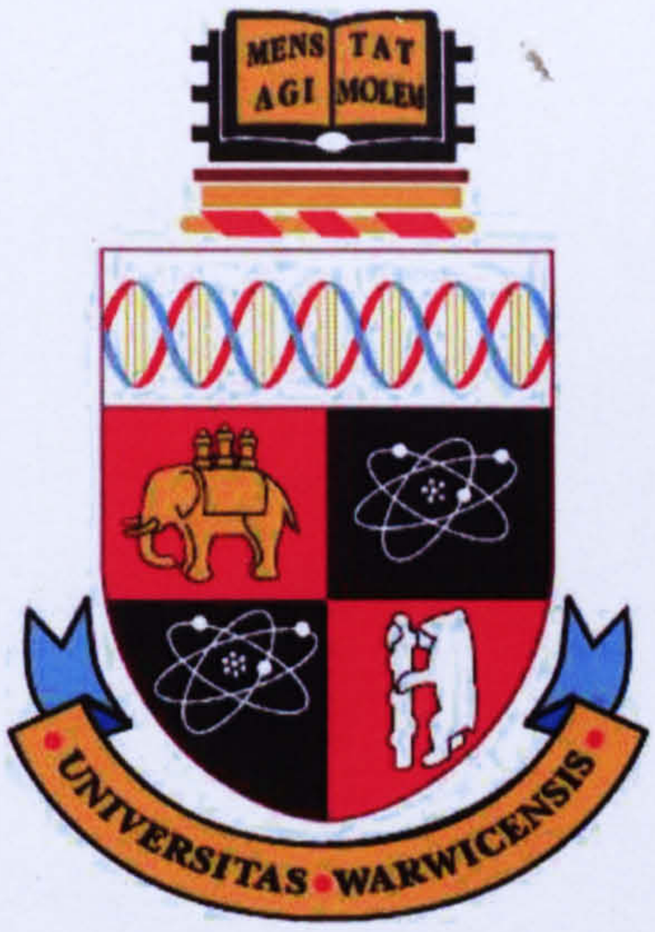

Self-adaptive Grid Resource Monitoring and Discovery

by

Hélène Niuklan Lim Choi Keung

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

January 2006

THE UNIVERSITY OF
WARWICK

Contents

List of Figures

List of Tables

Acknowledgments

This thesis describes work carried out at the High Performance Systems Group (HPSG) in the Department of Computer Science, the University of Warwick, between October 2001 and September 2004. This work was made possible by funding through the Warwick Postgraduate Research Fellowship.

There are several people who have provided support and assistance throughout this work. I would first like to thank my supervisor, Dr. Stephen Jarvis for his guidance and encouragement, and for making this work a rewarding experience. My thanks also go to my advisor, Prof. Graham Nudd for his sound advice on many aspects of research. I would also like to thank the members of the High Performance Systems Group for their help at all stages of this work.

I also wish to express my thanks and appreciation to my family for their unfaltering support and understanding. Last but not least, I would like to thank my husband and best friend Justin, for his continuous love, support and encouragement.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

This thesis was typeset with LaTeX 2_ε[1] by the author.

The elements of performance arising from the Monitoring and Discovery Service 2 have been published in [67, 68, 69]. The integration of MDS2 with other Grid components has been published in [72, 66, 104, 53, 55, 54]. The OGSI-related benchmarking research has been published in [70], while features of GridAdapt concerning self-adaptation and optimisation have been published in [71].

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society.

Abbreviations

API	Application Programming Interface
DIT	Directory Information Tree
DOM	Document Object Model
DN	Distinguished Name
GASS	Global Access to Secondary Storage
GGF	Global Grid Forum
GIIS	Grid Index Information Service
GIS	Grid Information Service
GLUE	Grid Laboratory Uniform Environment
GMA	Grid Monitoring Architecture
GRAM	Globus Resource Allocation Manager
GridFTP	Grid File Transfer Protocol
GRIS	Grid Resource Information Service
GSH	Grid Service Handle
GSI	Grid Security Infrastructure
GSR	Grid Service Reference
HTTP	HyperText Transfer Protocol
LDAP	Lightweight Directory Access Protocol
LDIF	LDAP Data Interchange Format
MDS	Metacomputing Directory Service
MDS2	Monitoring and Discovery Service
MDS3	Monitoring and Discovery System

MJS	Managed Job Service
NIC	Network Interface Card
OGSI	Open Grid Service Infrastructure
OID	Object Identifier
PHP	Hypertext Preprocessor
QoS	Quality of Service
R-GMA	Relational Grid Monitoring Architecture
RFT	Reliable File Transfer
RIPS	Resource Information Provider Service
SDE	Service Data Element
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	Time-to-live
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VO	Virtual Organisation
WS	Web Services
WSDL	Web Service Description Language
WSRF/WSN	Web Services Resource Framework/Web Services Notification
XDR	eXternal Data Representation
XML	Extensible Markup Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations

Abstract

The Grid provides a novel platform where the scientific and engineering communities can share data and computation across multiple administrative domains. There are several key services that must be offered by Grid middleware; one of them being the *Grid Information Service* (GIS). A GIS is a Grid middleware component which maintains information about hardware, software, services and people participating in a *virtual organisation* (VO). There is an inherent need in these systems for the delivery of reliable performance. This thesis describes a number of approaches which detail the development and application of a suite of benchmarks for the prediction of the process of resource discovery and monitoring on the Grid. A series of experimental studies of the characterisation of performance using benchmarking, are carried out. Several novel predictive algorithms are presented and evaluated in terms of their predictive error. Furthermore, predictive methods are developed which describe the behaviour of MDS2 for a variable number of user requests. The MDS is also extended to include job information from a local scheduler; this information is queried using requests of greatly varying complexity. The response of the MDS to these queries is then assessed in terms of several performance metrics.

The benchmarking of the dynamic nature of information within MDS3 which is based on the *Open Grid Services Architecture* (OGSA), and also the successor to MDS2, is also carried out. The performance of both the pull and push query mechanisms is analysed. *GridAdapt* (Self-adaptive Grid Resource Monitoring) is a new system that is proposed, built upon the Globus MDS3 benchmarking. It offers self-adaptation, autonomy and admission control at the Index Service, whilst ensuring that the MDS is not overloaded and can meet its quality-of-service, for example, in terms of its average response time for servicing synchronous queries and the total number of queries returned per unit time.

Chapter 1

Introduction

1.1 Research Context

In recent years, distributed computing has entered a new phase where advanced networking facilities link up machines with considerable computational power and sophisticated data storage methods. This fact coupled with the growing demands from user applications has led to the creation of *Grid Computing* which is the dynamic use of potentially unlimited resources.

The Grid provides a platform where the scientific and engineering communities can share data, resources, people and computation across multiple administrative domains. This infrastructure provides large-scale pooling of resources, whether compute cycles, data, sensors, or people [32, 34]. To allow such pooling of resources, the necessary hardware infrastructure should be available to achieve the necessary interconnections, as well as the required software infrastructure to monitor and control the total aggregation of resources. Given below is a list which describes the fundamental characteristics of the Grid infrastructure:

- The Grid needs to provide *dependable* service in the form of predictable, sustained and high levels of performance from the various components of the infrastructure. It is only when this condition is satisfied that applications are written and used. Moreover, the relevant performance aspects include that from application to application,

network bandwidth, latency, jitter, computer power, software services, security and reliability.

- The service provided by the Grid should also be *consistent*. Those services should be accessible via standard interfaces and operating within standard parameters. One of the significant challenges imposed by the development of standards, is encapsulating heterogeneity without compromising high-performance execution.

- Grid services should also be *pervasive* as services must be accessible, irrespective of the environment the client is in. This feature of the Grid relies on the resources being available in a controlled fashion.

- *Inexpensive* access to Grid services is also important to ensure the long-term and widespread use of the Grid. The benefits of using the Grid should also outweigh the costs involved.

1.2 Research Motivation

There are several key services that must be offered by Grid middleware; one of them being the Grid Information Service. A *Grid Information Service* (GIS) is a Grid middleware component which maintains information about hardware, software, services and people participating in a virtual organisation (VO) [37]. Upon request from Grid entities, the GIS can launch resource discovery and lookup [21, 6]. Without this service, it would be very difficult for users to know which resources are available to use, especially when the resources are geographically dispersed.

Additionally, Grid environments create the implicit need for applications to obtain real-time information about the structure and state of the dynamic components of the meta-system, to utilise this information to make configuration decisions, and to be notified when information changes. To be able to offer these capabilities, the monitoring and discovery

infrastructure should be scalable, flexible, extensible, easily configurable and robust. It should also be able to support a very large number of resources which are constantly changing in status and availability.

Nevertheless, conflicting inherent requisites of these Grid discovery and monitoring systems are their sustained performance and fulfilling of quality-of-service requirements.

Currently, there are no mechanisms for regulating either the rate of arrival of synchronous queries or the rate of event notification, at the level of Grid Information Services. Consequently, such a service runs the risk of being overloaded and not being able to deliver its quality-of-service in terms of the average query response time, the freshness of the data and the number of queries serviced per unit time. There is therefore the requirement for the GIS to adapt automatically to the nature and number of queries at any one time, as well as meet its own targets in terms of the number of queries returned per unit time and the validity of the data in question.

The system proposed in this thesis, GridAdapt, achieves the above requirements by utilising performance benchmarks of the MDS and self-adapting and self-optimising to ensure continuous delivery of uptodate dynamic data. The methodology used is to run particular experiments a number of times, followed by statistical analyses to interpret the results. Results obtained from experiments are promising, and it is also envisaged that GridAdapt can be easily be mapped to other distributed environments where dynamic information is required, for instance, peer-to-peer systems.

1.3 Aims and Objectives

The main objective of this thesis is to address the problem of a GIS not being scalable with an increasing number of concurrent queries and users. This scalability is an important feature of the Grid as the GIS should be a dependable service where predictable and high levels of performance are offered. As the Grid involves a large number of resources with

dynamic data, this feature is of the utmost importance. The aims and objectives of the work in this thesis are therefore:

- To analyse and benchmark the performance of the MDS with an increasing number of concurrent queries;
- To analyse and contrast the information provision mechanisms of the MDS;
- To benchmark both the push- and pull-based query types of the MDS;
- To develop self-adaptive algorithms to improve the scalability of the MDS;
- To demonstrate that the proposed self-adaptive system is more dependable than the existing MDS.

1.4 Thesis Contributions

The contributions of this thesis include:

- A comparative performance study of MDS2 (Metacomputing Directory Service) queries at both the local levels, with different back-end implementations, and the effect on Grid applications. The difference in scalability from three monitoring and information systems is studied in [129]. In this thesis, the focus is on the difference or similarity in querying an information server for the same set of information, based on the varying information gathering methods at the source. Moreover, the contribution which we make addresses comparable issues as in [129]; for instance, the effect network behaviour has on the performance observed, especially, the load of the information server. All these performance studies are crucial in proposing recommendations for the deployment of the monitoring and information systems under consideration. Previous work contributing to the earlier MDS versions include [3, 101].

- The performance prediction of MDS2 for global level queries and comparison of various predictive algorithms.

- The integration of the Titan [104] scheduler with the MDS2 to allow for the publication of job information.

- The performance analysis of queries with varying complexity for the MDS2 to test its scalability with dynamic information.

- The benchmarking of the MDS3 push and pull query mechanisms under varying query loads, and the analysis of client-side performance.

- The development of push self-adaptive algorithms based on the characterisation and recommendations gathered from the benchmark data.

- The formulation of a pull self-adaptive algorithm and admission controller to ensure server-side stability.

- A novel self-adaptive Grid Information Service which gives a performance improvement with different workloads. While research has been previously carried out to test the performance of GIS and queries, to the author's knowledge, no other work has been carried out to regulate their operation automatically.

1.5 Thesis Outline

This thesis includes the development of a novel self-adaptive and self-optimising Grid resource monitoring system, *GridAdapt* which offers performance improvement to users querying the Index Service, as well as to the MDS itself. GridAdapt aims to achieve a balance between the number of push and pull queries, while maintaining QoS aspects including the average query response time and the number of queries serviced by the Index Service in a given time period. Benchmarks of the MDS3 are first compiled and

the data characterised before the self-adaptive algorithms and the admission controller are derived. The first part of the thesis consists of a detailed performance analysis of querying the MDS2 at local levels, using various information provision mechanisms. The second part of the MDS2 work comprises the integration of data from the Titan scheduler job execution management, and the querying of that data using disparate queries. The thesis is organised as follows:

Chapter 2 introduces Grid Computing, its architecture, its evolution and current middleware. An overview and critical analysis of several solutions to the distributed resource monitoring and discovery issue are also described.

Chapter 3 details the architecture and components of both the Globus MDS3 and MDS2. The MDS is chosen as the basis for GridAdapt due to the popularity and widespread deployment in production Grids.

Chapter 4 introduces related software from the High Performance Systems Group at the University of Warwick. It also presents the benchmarking of the performance of the MDS2 with different information provision mechanisms, and discusses performance prediction algorithms which can be derived from the benchmark data. The addition of scheduler information to the MDS is also discussed, as well as the performance of disparate queries on this information.

Chapter 5 presents the benchmarking of the MDS3 for both push and pull query mechanisms, and the results are analysed.

Chapter 6 describes further performance studies on the MDS3 for concurrent push and pull queries. The self-adaptive algorithms used in GridAdapt are developed and its performance benefits identified.

Chapter 7 draws conclusions from the work presented in this thesis and suggests possible enhancements to the algorithms and features of GridAdapt.

Chapter 2

Grid Middleware

2.1 Introduction

The term *Grid* defines a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities. A wide range of Grid computing organisations are working towards the interoperability of Grid systems and to accomplish the main objectives of this distributed computing paradigm. This chapter includes details of the Grid architecture, the evolution of Grid computing, and an overview of the Globus Toolkit which provides the core framework for the realisation of a computational Grid. Each of the various current Grid discovery and monitoring systems is described and a critical analysis is also presented. This chapter concludes with an overview of several relevant Grid resource management systems.

2.2 Grid Architecture

The Grid infrastructure enables collections of resources to be managed independently, but also cooperatively, and accessible to a user community. Sharing occurs amongst those resources which include computers, software and data. Additionally, the Grid allows collaborative, problem-solving and resource brokering, to take place. This level of sharing is highly controlled, with resource providers and consumers clearly defining the resource

to be shared, and the conditions for the occurrence of such sharing. The collection of resources and organisations governed by these sharing rules, is called a *virtual organisation* (VO) [37, 34]. VOs therefore allow disparate groups of organisations and individuals to share resources in a controlled manner, so that members can collaborate to achieve a shared goal.

Furthermore, the Grid infrastructure is required for establishing, managing and exploiting dynamic, cross-organisational sharing relationships which make VOs possible. This Grid infrastructure identifies fundamental system components, specifies the purpose and function of these components, and define the types of interactions these components have with one another. Being an extensible, open architectural structure, the Grid allows additional components to be added to meet VO requirements. Moreover, components can be organised into layers, as shown in Figure 2.1, with each layer sharing common characteristics and also building on capabilities and behaviours provided by each relative lower layer.

The basis of the Grid architecture is the principle of the *hourglass model* [95], where the narrow neck at the centre of the hourglass defines a small set of core abstractions and protocols. The top of the hourglass represents a number of different high-level behaviours which map onto the protocols. Additionally, the base of the hourglass represents underlying technologies onto which the protocols map. The core set of protocols is small by definition, and provides *resource* and *connectivity* capabilities.

As shown in Figure 2.1, the Grid infrastructure can be represented as having the following layers: *fabric*, *resource*, *connective* and *collective*.

- **Fabric** This layer provides resources, including compute nodes, storage systems, catalogues, networks and sensors. These resources could be logical entities, for example, a computer cluster, or a distributed file system, but this is not the concern of the Grid architecture. The fabric components are responsible for implementing

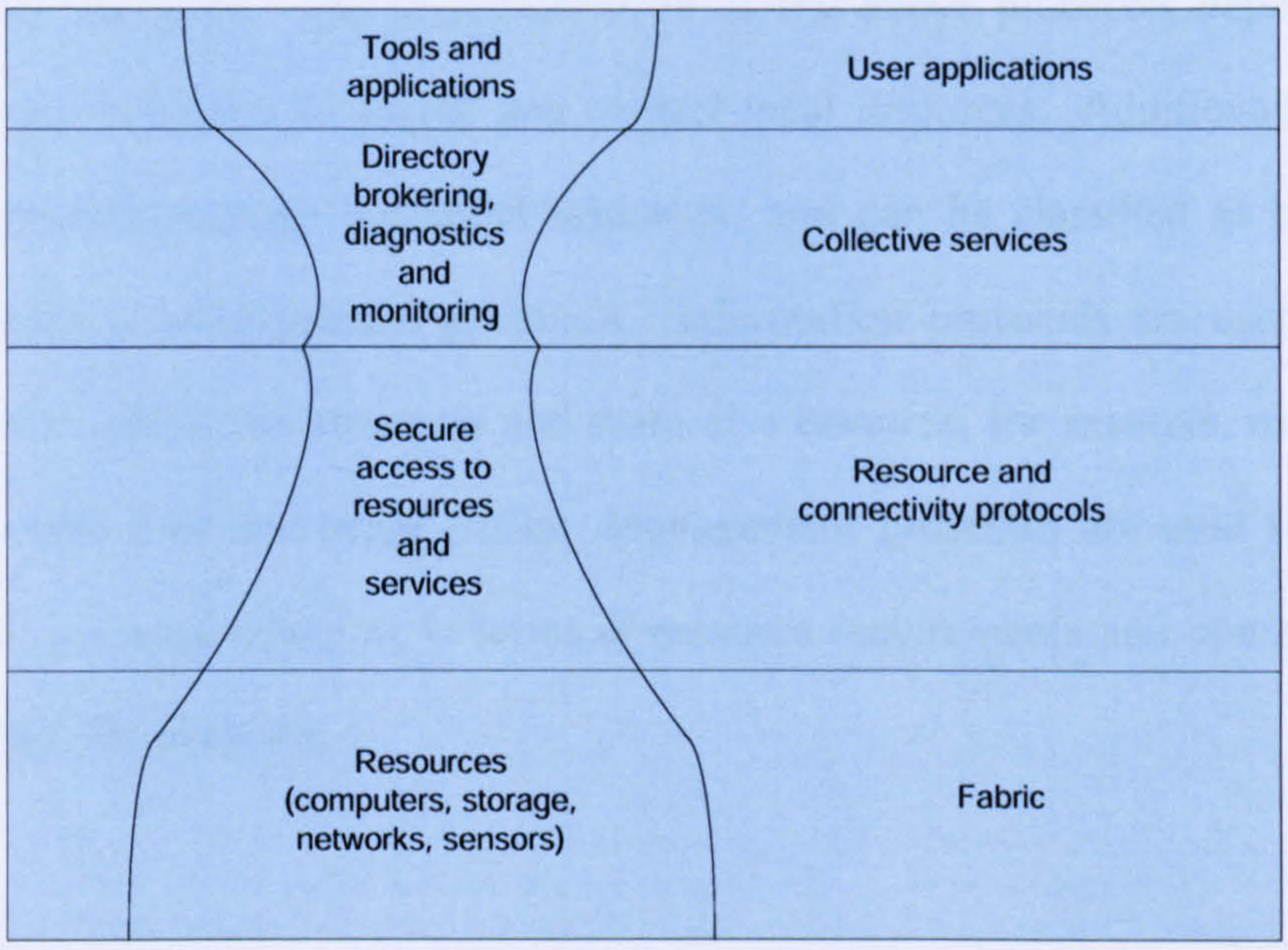

Figure 2.1: Components of the layered Grid architecture.

local, resource-specific operations which are required on specific, physical or logical resources. These operations in turn, enable sharing operations at higher levels.

- **Connective** The core communication and authentication protocols needed for Grid-specific network transactions, are defined in the connectivity layer. Those communication protocols enable the exchange of data amongst fabric layer resources. On the other hand, being built on communication services, authentication protocols provide cryptographically secure mechanisms for verifying the identity of users and resources. It is also important that connectivity layer security solutions be based on existing standards, and provide flexible support for communication protection. For example, there should be control over the degree of protection, support for reliable transport protocols other than TCP and stakeholder control over authorisation decisions.

- **Resource** Building on the communication and authentication protocols within the connectivity layer, the resource layer defines protocols for the secure negotiation,

initiation, monitoring, control, accounting and payment of sharing operations on individual resources. The implementation of the above protocols depends on the fabric layer functions to access and control local resources. Additionally, resource layer protocols manage individual resources, and can be classified as being either *information* or *management* protocols. Information protocols are used to obtain information about the structure and state of a resource, for example, its configuration, current load and usage policy. Management protocols are used to negotiate access to a shared resource, in terms of resource requirements and operations to be performed, for instance.

- **Collective** This layer contains protocols and services which are used to capture the interactions across collections of resources. The collective layer does not therefore manage individual resources, but instead, build upon the small set of protocols in the resource and connectivity layer in the "hourglass", to implement a wide variety of sharing behaviours. At the same time, no new requirements are placed on the resources being shared. Examples of collective layer services are directory services, monitoring and diagnostic services, co-allocation and brokering services, and data replication services. Moreover, programming models and tools both define and invoke collective layer functions; instances of these models and tools are workflow systems, software discovery services and collaboratory services. Furthermore, security, policy and accounting issues are addressed in the collective layer services. The collective layer protocols range from general purpose to highly application- or domain-specific. Collective functions can also be implemented as standalone services or as libraries designed to be linked with applications. It is also possible for collective components to be customised to the requirements of a specific user community or application domain. Additionally, if the target user community is large, the collective components should be based on standard protocols and APIs.

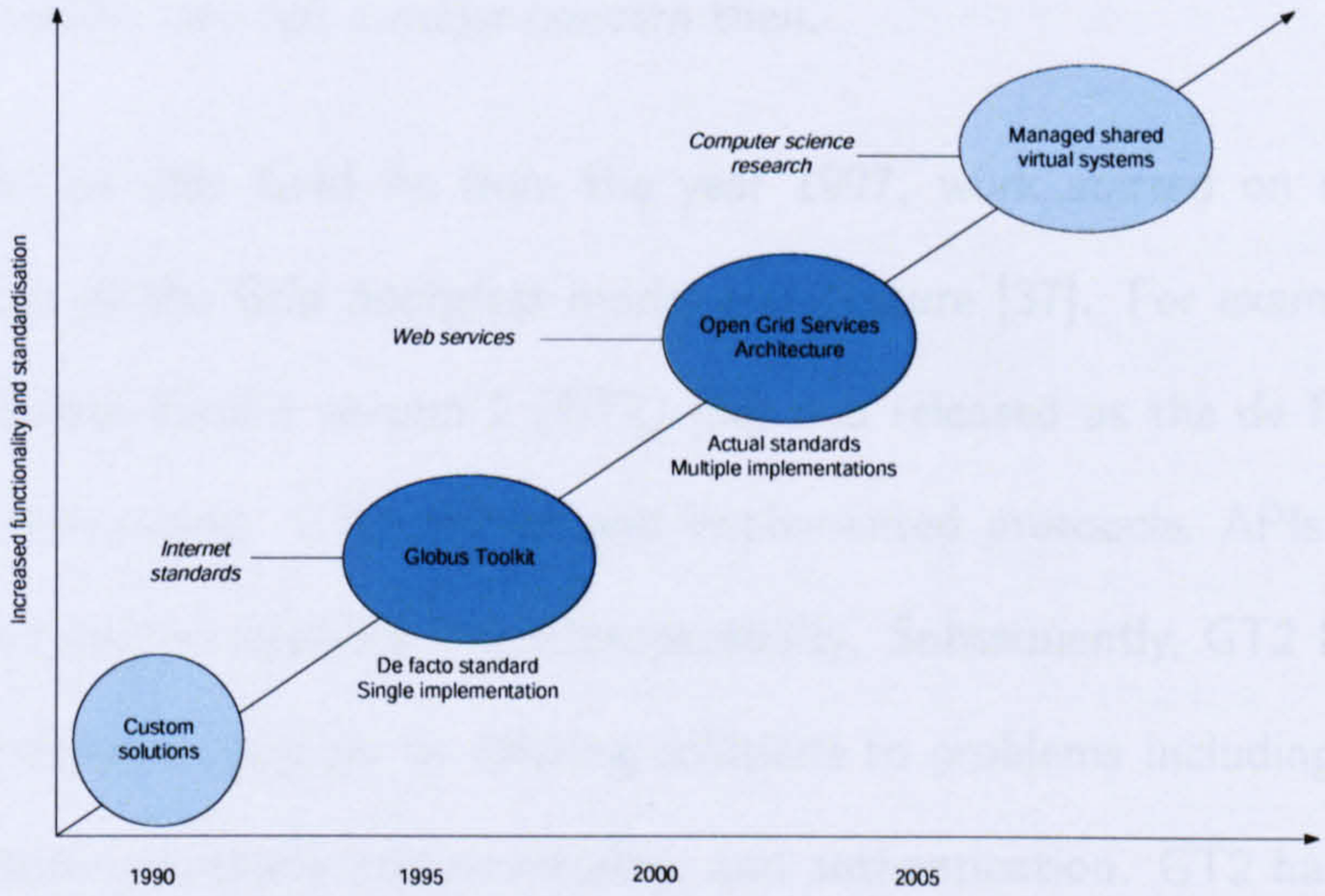

Figure 2.2: The evolution of Grid technologies.

2.3 The Evolution of Grid Computing

Over the last decade, major changes have taken place in the way businesses, global industries and individual users utilise computing devices, as a result of significant research in both academia and industry. The emphasis has shifted from localised computing resources and services to environments supporting widely distributed and complex services. Traditional computer systems consisted of monolithic and non-interoperable solutions but with the advent of Grid computing, solutions concentrated on large-scale resource discovery, utilisation and sharing within virtual organisational boundaries. Currently, applications and solutions for Grid computing are open technology-based and service-oriented. The evolution of Grid technologies [34] can be staged into four distinct phases, as illustrated in Figure 2.2.

1. **Custom solutions** Solutions to Grid computing problems in the early 1990s, were custom-developed. New ideas about the possibilities of metacomputing [14] were being tested out. Built directly on Internet protocols, applications possessed lim-

ited functionality in terms of security, scalability and robustness. The concept of interoperability was not a major concern then.

2. **Anatomy of the Grid** As from the year 1997, work started on several implementations of the Grid *hourglass* model architecture [37]. For example, the open source Globus Toolkit version 2 (GT2) [33] was released as the de facto standard for Grid computing. GT2 defined and implemented protocols, APIs and services, whilst focusing on usability and interoperability. Subsequently, GT2 has supported Grid deployments globally by offering solutions to problems including resource access, resource discovery and monitoring, and authentication. GT2 has also allowed the rapid development of Grid applications with the help of standard protocols and services, and through the leverage of existing Internet standards for transport, resource discovery and security. Moreover, the GT2 protocol suite was reviewed by standards bodies, and a number of technical specifications produced. Nevertheless, GT2 standards were neither formal nor subject to public review.

3. **Open Grid Services Architecture** Being a real community standard with multiple implementations, the Open Grid Services Architecture (OGSA) [36] emerged in 2002, with the OGSA-based GT3 being released in 2003. OGSA builds on GT2 concepts and technologies, and fully supports industry efforts in service-oriented architecture and Web services. Furthermore, OGSA defines a core set of standard interfaces and behaviours, as well as a framework for the definition of a large range of interoperable, portable services.

4. **Managed, shared virtual systems** Grid visionaries believe that the definition of the initial OGSA technical specifications is a major step forward towards the realisation of the Grid. The future will see a growing set of interoperable services and systems, which possess higher degrees of virtualisation, more types of sharing and increased qualities of service, all being built upon OGSA's service-oriented infras-

tructure. Moreover, increasing support from industry will be available, leading to very recent Grid implementations such as GT4. Being based on the Web Services Resource Framework (WSRF) [19, 20], GT4 is an implementation of a set of six Web services specifications which define a *WS-Resource approach* to modelling and managing state in a Web services context. Proposed in January 2004, the WS-Resource Framework aims to exploit new Web services standards by re-factoring the Open Grid Service Infrastructure (OGSI). WSRF deals with the creation, addressing, inspection and lifetime management of stateful resources. Additionally, the framework allows state to be expressed as stateful resources and the relationship between Web services and stateful resources to be specified in terms of Web services conventions.

With the immense on-going research and effort in Grid computing, global standardisation is crucial for the efficient, widespread application of the concept. Therefore, worldwide communities of users, developers and vendors exist to lead such effort. For instance, the *Global Grid Forum* (GGF) [40] community consists of thousands of people in both industry and research in more than 50 countries, working in community-initiated groups to develop best practices and specifications. This work is carried out in collaboration with other leading standards organisations, software vendors and users. Additionally, the GGF aims to create an international community for the exchange of ideas, experiences and requirements.

Another organisation involved in the standardisation of Grid standards is the *Distributed Management Task Force* (DMTF) [23] which is leading the development of management standards and integration technology for enterprise and Internet environments. In partnership with major technology vendors and standards groups, DMTF approaches management in an integrated and cost-effective manner, through interoperable management solutions. To provide common management infrastructure components for instrumentation, control and communication, DMTF leverages technologies including the Common Information

Model (CIM), communication and control protocols such as Web-based Enterprise Management (WBEM), the Systems Management Architecture for Server Hardware (SMASH) initiative and core management services. Moreover, the *Storage Networking Industry Association* (SNIA) [105] is involved in driving storage industry standards, best practices and education, while the *Organisation for the Advancement of Structured Information Standards* (OASIS) [86] is a non-profit international consortium for the development, convergence and adoption of e-business standards.

2.4 The Globus Toolkit

Being a multi-institutional research effort, the *Globus* project [109] aims to develop a basic infrastructure for the realisation of a computational Grid, as well as higher-level services for its management and control. After numerous development iterations, the Globus Toolkit now offers the possibility of increasing the average and peak computational performance which is available to applications, irrespective of the spatial distribution of both resources and users.

Figure 2.3 illustrates the Globus infrastructure, with a specific reference to the Globus GT3 release. All of the services (MDS, GRAM and GIS) shown are also found in GT2. Providing a layered software architecture, Globus caters for low-level services to high-level ones. These high-level services offer resource discovery, monitoring, allocation, security, data management and access. The Core layer provides a framework for the high-level services which include the *Globus Resource Allocation Manager (GRAM)*, the *Grid Security Infrastructure (GSI)* and the *Grid Information Services (GIS)*.

GRAM acts as a single, standard interface for the request of the execution of jobs on remote system resources. GRAM is mainly used for facilitating remote job submission and control on remote resources. However, it also provides services ranging from resource allocation, process creation, monitoring to management. While GRAM does not provide

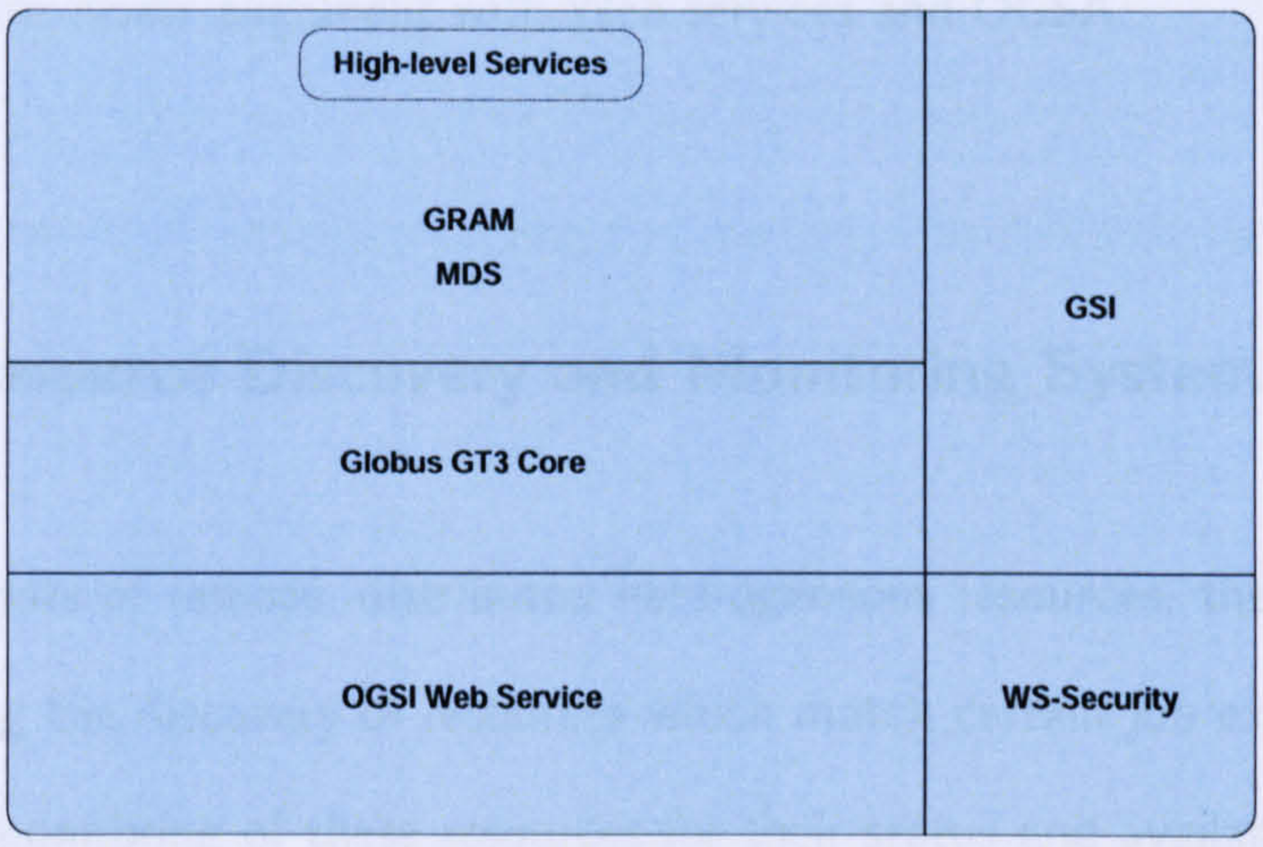

Figure 2.3: The components of the Globus GT3 offering middleware, core and high-level services.

resource brokering or scheduling capabilities, it utilises a high-level Resource Specification Language (RSL) to specify commands and to map them to the local schedulers and resources.

The Grid Security Infrastructure provides a single sign-on, authentication service which supports local control over access rights as well as mapping from global to local user identities. Being an authentication service able to run on any resource, GSI in GT3 maintains the previous mechanisms whilst being compatible with Web service security standards via a GSI profile for WS-Security [103].

A Grid Information Service provides information about Grid resources which is used in resource discovery, selection and optimisation. The earlier version of the Globus GIS, the Metacomputing Directory Service (MDS), is an extensible Grid service which offers data discovery mechanisms, based on the Lightweight Directory Access Protocol (LDAP). Building on a uniform framework, the MDS provides access to heterogeneous systems, as well as information about system configuration and status information including computer server configuration, network status, and the locations of replicated datasets. Additionally,

the GT3 framework integrates the MDS with the eXtensible Markup Language (XML) data framework for closer alignment with Web services and OGSA.

2.5 Grid Resource Discovery and Monitoring Systems

As the Grid consists of remote, distributed heterogeneous resources, there is a need for services, including the discovery of resources which match certain job execution requirements, and the monitoring of these resources for their status and availability. A number of software tools have been developed to meet those requirements, each of which provides specific features. However, none of these software tools offers all of the services required for locating and monitoring resources in Grid environments. The following sections present some of the most widely used resource discovery and monitoring systems, as well as a critical analysis of the characteristics which may be lacking.

2.5.1 Globus MDS

The Monitoring and Discovery System (MDS) [43] is the information services component of the Globus Toolkit and it provides access to information about the resources which are available for sharing in a Grid environment. It provides information about the status of the resources, which can be a compute node, a data storage or a network link. At the time of writing, there are three versions of the MDS available: MDS2, MDS3 and MDS4. While MDS2 is an LDAP-based implementation of information services, developed with the Globus Toolkit 2, MDS3 is an implementation based on OGSI and has been developed with the Globus Toolkit 3.0 and 3.2. Additionally, released with the Globus Toolkit 4.0, MDS4 is based on WSRF (Web Services Resource Framework). More detailed information on the MDS is given in Chapter 3.

2.5.2 GrADS

The *GrADS* project [5] is working towards the simplification of the use of distributed, heterogeneous computing. It is attempting to solve the scientific and technical problems that prevent the Grid from being used easily for application development and application performance tuning. Moreover, the project is focusing on four key areas, each of which produces a prototype infrastructure intended to facilitate Grid programming. These areas are Grid software architectures to facilitate information flow and resource negotiation; base software technologies including scheduling, resource discovery and communication; the development of Grid applications using programming languages, compilers and environments; and the development of mathematical and data structure libraries for Grid applications. Additionally, the GrADS project is involved in developing MicroGrid testbeds to carry out extensive tests as to the effectiveness of the technologies developed, whereas the evolving Grid testbeds are utilised for large-scale experimentation and demonstration. The project is also carrying out technology transfer on two fronts, namely the understanding of existing applications to be run on the Grid, and the collaboration with industrial partners to promote the standardisation and adoption of research-based system software technologies.

Program Preparation and Execution System

The GrADS project is also working on a software architecture to support adaptation and performance monitoring in dynamic distributed environments. This architecture called *GrADSoft*, proposes to adapt applications continuously to Grid conditions and particular problem instances, instead of following the sequence of application creation, compilation, execution and post-execution analysis.

The GrADSoft architecture contains an enhanced execution environment which leads to the highest level of overall performance by continually adapting the application to changes

occurring in Grid resources. Two inherent requirements are the encapsulation of applications as configurable objects and the existence of performance contracts. Another component of the architecture is the real-time monitor which provides information about the system performance through real-time displays and program development tools. This feedback information allows the user to steer program behaviour and make recommendations for the reoptimisation of specific components.

GrADS MacroGrid Testbed and Services

The *GrADS MacroGrid* is a large-scale, wide-area Grid application execution environment, which is required to verify MicroGrid results and to extend them beyond a MicroGrid environment. Moreover, the various GrADS institutions are collaborating for the integration of testbeds for carrying out controlled experiments on top of the GrADSoft architecture.

GrADS MicroGrid Toolkit

This toolkit is used to emulate different Grid configurations and dynamic behaviours in controlled environments, allowing computer scientists and application scientists to experiment systematically with various Grid configurations and behaviours.

The MicroGrid environment allows resource-controlled studies to be carried out for the characterisation of Grid software in terms of system stability, performance stability and robustness. It therefore helps to deploy full-scale dynamic systems, as well as build robust Grid applications.

GrADS Applications

The GrADS project at the University of Tennessee is focusing on the development of libraries and algorithms to harness dynamic, distributed and parallel environments, in an efficient and reliable manner. It is also responsible for the validation of these libraries and

algorithms when scientific and engineering applications are executed.

Some of the specific issues that have to be addressed, have been dealt with. For example, the GrADS software is able to manage both communication and the memory hierarchy, by combining both compile-time and run-time techniques. This allows current high-end machines to used effectively. New ways of building libraries are being thought of within the GrADS project as the Grid environment complicates issues of computation, memory hierarchies, latency ranges and run-time variability. Furthermore, those libraries and algorithms are being made Grid-aware by the introduction of parameters and the annotation with performance contracts. Additionally, new algorithms are being developed which use adaptive strategies by interacting with other GrADS components. For instance, libraries contain performance contracts for allowing for the dynamic negotiation of resources and the adaptive run-time support for the compiler, scheduler and runtime system.

GrADS Resource Selector

The *Resource Selector* [117] is a component of GrADS, which is used to obtain a list of available resources in the testbed. The Resource Selector accesses the Globus Monitoring and Discovery Service (MDS) for that purpose, before contacting the *Network Weather Service* (NWS) [112, 124, 126, 123] for detailed system information for these discovered resources. The GrADS application manager then contacts the *Performance Modeller* which uses a custom-built execution model built for the application, problem parameters and machine information to decide the final list of machines on which the application will be executed. Moreover, contracts are also considered and approved, thereby generating a final list of machines which is passed on to the Application Launcher. Furthermore, the GrADS architecture consists of a *GrADS Information Repository* (GIR) that stores the various states of the application manager and the numerical application. The GIR is also used to communicate various application states, for example completed or suspended jobs,

to the application manager. If the job is completed, the application manager exits and returns success parameters to the user. However, if the application stops, the application manager waits upon a resume signal before restarting the resource selection phase.

Resource Selector Service (RSS)

The *Resource Selector Service* (RSS) [74] is used to select Grid resources which are appropriate for the execution of a particular job, depending on the requirements for that job. It also organises the chosen Grid resources into a virtual machine with the correct topology and also supports the mapping of the application workload to the virtual machine resources. The three steps, *selection*, *configuration* and *mapping* can be viewed as a single process because the selector can only start comparing selections for the best one, after a mapping has been established.

The Resource Selector Service extends the Condor *ClassAds Language* [16] by allowing users to specify aggregate resource properties. Moreover, the RSS contains an extended *set matching* matchmaking algorithm which supports one-to-many matching of set-extended ClassAds with resources. Furthermore, the RSS enables both application resource requirements and application performance models to be specified declaratively, in the ClassAds language. Mapping strategies are specified by user-supplied code. In brief, the Resource Selector Service offers a general-purpose resource selection framework that can be interfaced by different types of applications.

A successful match in set matching is defined as occurring between a single *set request* and a *resource set*. The set request is expressed in set-extended ClassAds syntax, which is identical to that of a normal ClassAd. However, the set-extended ClassAd syntax can represent both *set expressions* and *individual expressions*. While set expressions place constraints on the collective properties of an entire resource ClassAd set, individual expressions apply singly to each resource in the set. The set-matching algorithm which the

RSS uses, evaluates a set-extended ClassAd request against a set of resource ClassAds, and it constructs a resource set which satisfies both individual and set constraints. The highest-ranking resource set is returned if the set match is successful.

The set-matching algorithm consists of two phases. In the *filtering* phase, individual resources are removed from consideration, based on individual expressions in the request. In the *set construction* phase, the set-matching algorithm attempts to discover a resource set that is most appropriate for the application.

The Resource Selection Service is a general-purpose framework, based on the above set-matching technique. It uses the Grid Information Service to obtain resource information, and selects the highest-ranking resource set, based on user resource requests. Moreover, the framework provides an open interface which users can utilise to customise the resource selector by specifying the application-specific mapping module. The Grid Information Service comprises the Monitoring and Discovery Service (MDS2) and the Network Weather Service (NWS) which both allow distributed resources to be discovered, accessed and periodically monitored. Furthermore, there are three modules in the Resource Selector Service. Firstly, the *resource monitor* acts as a Grid Resource Information Service (GRIS) [21] and it queries the MDS and NWS for resource information which it caches in local memory. Secondly, the *set matcher* uses its algorithm to match incoming application requests with the best set of available resources. Lastly, the *mapper* is responsible for deciding the topology of the resources and mapping the workload of the application to resources.

2.5.3 Network Weather Service

The goal of the *Network Weather Service* (NWS) is to provide accurate forecasts of dynamically changing performance characteristics from a distributed set of metacomputing resources. It therefore performs active monitoring in a Grid environment by using a collection of monitoring servers which measure network latency and bandwidth between

pairs of nodes. These observations are subsequently used in future predictions of network performance. Moreover, the NWS contains a family of predictors and it dynamically selects a prediction function based on observed throughput and latency.

Moreover, the NWS forecasting engine applies mathematical models to a series of measurements which are taken at the application level to ensure that predictions are close to the performance available to applications [38]. The NWS takes these steps to generate forecasts. Firstly, it operates several different models simultaneously and computes a forecast from each of them. At the next time step, when a measurement is taken, it is compared to each forecast and the subsequent forecasting error is recorded for each model. When the NWS later receives a forecast request, the forecasting engine examines the cumulative error measures recorded for each forecasting model, and it selects the one showing the least cumulative error up to that point in time, to generate the forecast. In brief, the NWS provides a ubiquitous service which can both monitor dynamic performance changes and remain stable itself. To achieve this objective, the NWS requires adaptive programming techniques, an architectural design supporting extensibility, and internal abstractions which can be implemented efficiently and portably.

The NWS produces short-term performance forecasts based on historical performance measurements. Its goal is to dynamically characterise and forecast the performance deliverable at the application level from a set of network and computational resources. These forecasts have been successfully used for applications, including dynamic scheduling agents for metacomputing applications and a selector of replicated web pages.

The operation of the NWS in different metacomputing and distributed environments, has provided insight into adaptive programming techniques, distributed fault-tolerant control algorithms and an extensible architecture. Furthermore, the design of the NWS aims to maximise four potentially conflicting functional characteristics. Firstly, the NWS should provide accurate predictions of future resource performance in a timely manner. Secondly,

the system must be non-intrusive and must therefore impede on the resources it is monitoring as little as possible. Thirdly, the NWS must have execution longevity, that is, it must be continually available as a general system service. Lastly, it should be ubiquitous in that it must be available from all potential execution sites within a resource set. It should also be accessible to all resources for monitoring and forecasting.

Moreover, the NWS consists of four different *component processes*. The *persistent state* process stores and retrieves measurements from persistent storage. The *name server* process provides a directory capability to bind process and data names with low-level contact information. Furthermore, the *sensor* process collects performance measurements from particular resources. As for the *forecaster* process, it predicts values for deliverable performance for a specific resource, during a specified time frame.

All NWS processes are designed to be nameless so that the overall system is more robust. Persistent state is handled explicitly by *Persistent State* processes throughout the whole system. A Persistent State process offers a text string storage and retrieval service and associates each stored string with an optional time stamp. The name of the data set that is to be accessed, should accompany each storage or retrieval request. Similarly, data which is sent to a Persistent State process should be immediately written to disk before an acknowledgement is returned. Moreover, since the forecasts generated by the NWS are useful only temporarily, the files used by the Persistent State processes, are managed as circular queues. In an early implementation version, the NWS maintains its custom-written naming and directory service to manage name-location bindings. A name is represented as a human-readable text string and a location is a TCP/IP address and port number. However, all data is manipulated as text strings. Since the circular queue management techniques were proving inappropriate, the Naming Service was being re-implemented in the Lightweight Directory Access Protocol (LDAP).

Critical Analysis of the NWS

Clique grouping came about with the network traffic overhead caused by the measurement of bandwidth between each pair of Grid nodes. Cliques can greatly reduce the measurement costs by assigning machines to groups; these machines are typically on local administrative domains and have complete mutual connectivity and no significant wide-area bandwidth differences. Each clique has a representative node which is involved in the measurement of bandwidth between cliques.

The challenge in creating cliques is that the choice for the representative nodes is not immediately obvious, nor is the decision for the members of the clique. This difficulty is enhanced with the existence of high bandwidth wide area networks which result in local area machines having less mutual bandwidth. Subsequently, extensive bandwidth measurements are required to determine the representative node for a clique. Moreover, representing nodes have to be modified when faults and changes occur in the network. These problems can quickly become uncontrollable when network administrators have to monitor machines in a large Grid.

NWS and Grid Information Service

Previous research has been carried out with the aim of integrating dynamic performance information from the Network Weather Service (NWS) into the Grid Information Service infrastructure (GIS), more specifically, the Globus MDS2 [106]. The intended resulting service is a unified information service for Grid systems, which manages both static and dynamic data efficiently. Moreover, the implementation developed provides a uniform Grid information interface for users, enabling Grid schedulers to be supported. This is achieved by describing a new data model for dynamic Grid information and specifying the relation of the model to the NWS and major reference GIS implementations which are based mainly on the MDS. Additionally, a caching NWS server has been developed, which

binds the object model to LDAP syntax and integrates it with the MDS2 hierarchy.

The integration of the NWS with the MDS focuses on the issues revolving around data organisation, as compared to data presentation. Another concern is the efficient translation from the data organisation to any particular presentation. In this system, each datum is defined in the third normal form as an efficient implementation, where redundant information is minimised. This data representation allows the efficient, mechanical translation between objects from different systems where they have different representations.

The NWS data model consists of two types of objects: *registration* objects and *data* objects. Registration objects are used by the NWS to monitor the data and entities that it manages. Even though these classes are derived by the requirements of the NWS, the authors of the latter believe that those classes can be extended to provide a general object model to other monitoring systems managing dynamic data.

The purpose of registration objects is to optimise data access by particular applications or Grid programming tools. For instance, registration objects contain information about the storage location of data objects, the medium used to store data as well as replication information.

Moreover, the infrastructure being described, allows the NWS to be a stand-alone system. Nevertheless, the benefit of the service is the registration of the NWS with the MDS using the Grid Resource Registration Protocol (GRRP). Additionally, every registration object called *GridDaemon* should respond to a call to the Grid Information Protocol (GRIP) with an expected LDAP Data Interchange Format (LDIF) block. Furthermore, the system enables clients and other GIS servers to subscribe to events and create user-specified registration objects.

In order to achieve performance, the name server information is provided by LDAP instead of by the original APIs. A further step is to have the MDS as a name service for the NWS since the former provides a wide user community access to globally located resources.

Moreover, external tools, including the GrADS Resource Selector, are able to access internal, structural performance information via a single LDAP information interface.

Simultaneously, the MDS is treated as an extra component of the NWS reporting system, and is used to disseminate measurement and forecasting data. Previous experience with GrADS led to the development of a set of caching policies for the performance optimisation of NWS data. Therefore, the implementation of the MDS interface to the NWS is via a daemon process, called *NWSlapd*, which offers a caching LDAP interface to the NWS. The *NWSlapd* represents the point of registration for the various NWS subsystem processes. Moreover, this daemon process periodically refreshes cached data and since it is based on OpenLDAP, the *Slurpd* implementation can be used to provide replication. Subsequently, each Virtual Organisation can have its own *NWSlapd* to serve as a Grid Resource Information Service (GRIS) node in the MDS hierarchy.

Experimental results [106] show that the NWSlapd performs well both when the cache is empty and when it is non-empty. Additionally, the NWS LDAP daemon shows better performance than the native NWS interface when the number of hosts increases. This is due to the NWS LDAP daemon actively caching NWS registration information and the location of each datum being resolved automatically, thereby the redundancy of requesting locations explicitly.

Critical Analysis of NWS and GIS

Depending on the expected data format by clients of the NWS, the change in the MDS implementation language from LDAP to XML, may introduce inefficiencies in the system through inconsistent infrastructure implementation. It might also be the case that the data format to be used from the merger of the NWS and MDS is dictated by the implementation language of the MDS. While this step might be beneficial to the generic middleware, it should not interfere with the efficiency of the internal structure of the NWS.

2.5.4 Condor Classads

Condor offers a general resource selection framework, based on the *ClassAds language* [16] which allows users to describe resource requests and resource owners to characterise their resources. There is also a component of Condor, called the *matchmaker* [94] which matches user requests with appropriate resources. If multiple resources fulfil a request, a *ranking* mechanism is used to sort through the available resources according to criteria provided by the user. The most appropriate resource which matches the user's request is selected.

A ClassAd (Classified Advertisement) is used to describe jobs and resources so that the current state of the particular system can be discovered. A ClassAd is also a mapping from *attribute names* to *expressions*. The expressions can be simple constants or a function of other attributes. Moreover, a protocol exists for *evaluating* the attribute expression of a ClassAd with respect to another ClassAd. For instance, the expression `"other.size > 3"` in one ClassAd evaluates to **true** if the other ClassAd has an attribute named `size` and the value of that attribute is an integer greater than three.

Condor matchmaking takes two ClassAds and evaluates one with respect to the other. Two ClassAds are said to *match* if each ClassAd has an attribute `requirements` which evaluates to **true** in the context of another ClassAd. Additionally, a ClassAd can include an attribute named `rank` which evaluates to a numeric value representing the quality of the match. For the process of resource selection, the matchmaker compares a ClassAd request with every available resource ClassAd, and subsequently selects the highest-ranking resource which matches the request.

Nevertheless, the ClassAds mechanism is limited in that the matchmaker is designed to select a single machine to execute a job, and it cannot be used in the case of a job requiring multiple resources. Moreover, the Condor matchmaking system uses the *symmetric evaluation* approach where both requests and descriptions are expressed in the ClassAds syntax.

Critical Analysis of Condor ClassAds

Condor matchmaking can only deal with binary matches. The ClassAds mechanism is limited in that the matchmaker is designed to select a single machine to execute a job, and it cannot be used in the case of a job requiring multiple resources. *Gang matching* attempts to improve this aspect by enabling a ClassAd to specify multiple resources, but it does not support sets of resources defined by their aggregate characteristics. On the other hand, *set matching* allows resource sets to be specified via a ClassAd; however, this does not extend to multiple resources of different types. Additionally, with Condor ClassAds, it is difficult to query requirements information to the asymmetric representation of properties and requirements. This is where properties are defined by expressions and requirements are specified in a *requirements* statement. The way in which requirements are specified, also means that unfulfilled constraints for two descriptions which only fulfil their counterparts' requirements partly, become more difficult to identify.

2.5.5 Hawkeye ClassAds

Hawkeye [48] provides a lightweight approach for system administrators to monitor and manage distributed systems for automatic problem detection. Whilst still in its infancy and leveraging the technologies implemented by Condor and its ClassAds, *Hawkeye* provides sophisticated mechanisms for collecting, storing and using dynamic information about compute resources. Condor is an infrastructure for supporting high throughput computing by providing well-established mechanisms for monitoring and managing a set of jobs. *Hawkeye* can be used for monitoring various attributes within a group of systems and it can also contribute towards the management of these systems. Moreover, the configuration of *Hawkeye* is flexible due to its legacy from Condor.

Hawkeye can also be configured such that specified scripts are executed periodically. One

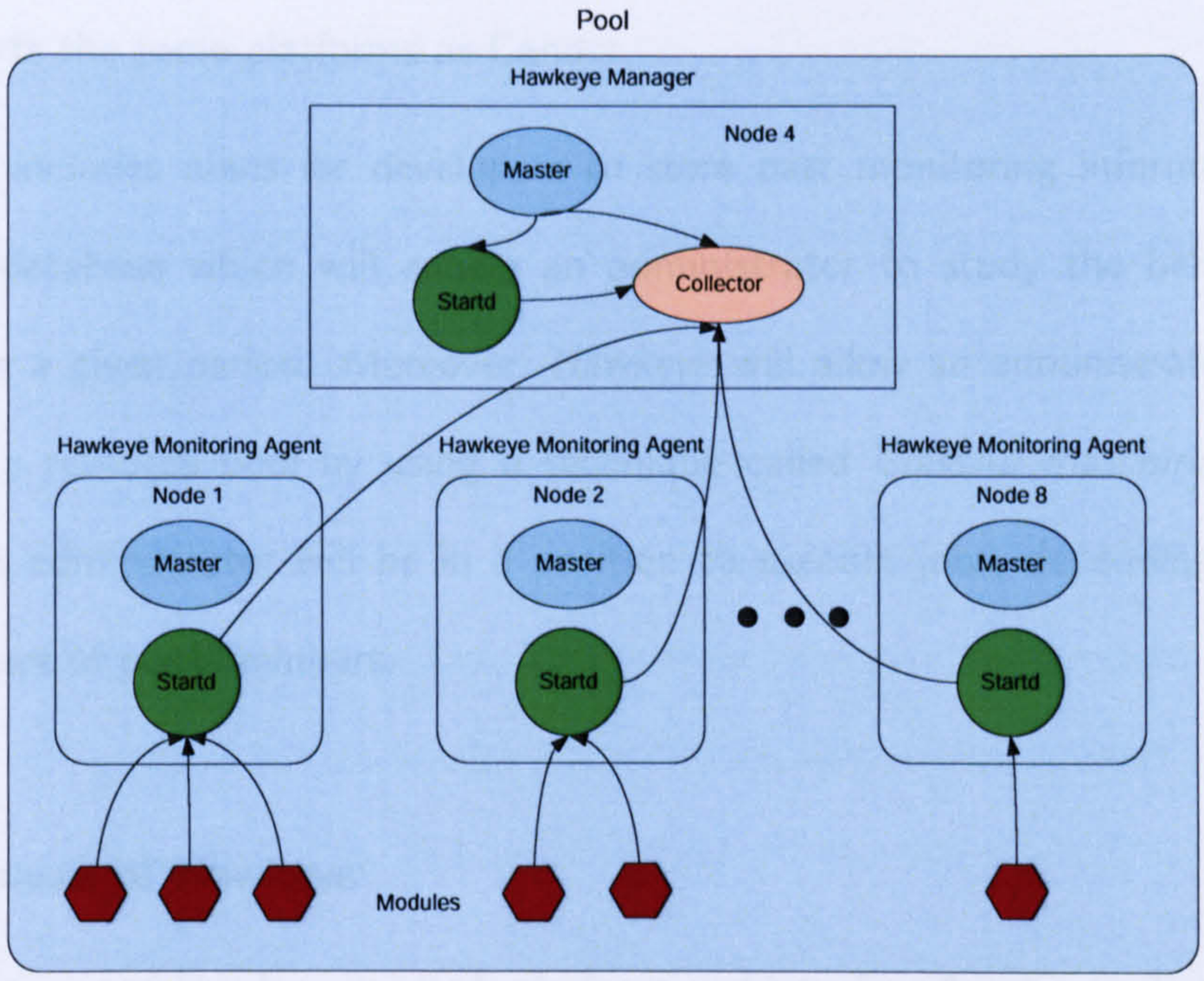

Figure 2.4: Components of the Hawkeye Architecture.

such script produces output in the form of a ClassAd attribute-value pair. Using defined naming conventions, these pairs are consequently added to the *machine* ClassAd. That machine ClassAd then contains attributes which can be useful in expressions including `START` and `SUSPEND` as well as the submit description file `REQUIREMENTS` expression.

As illustrated in Figure 2.4, the architecture of *Hawkeye* consists of four main components: the Hawkeye pool, Hawkeye manager, Hawkeye monitoring agent and Hawkeye module. The components are linked up in a hierarchical structure and can have several levels. Additionally, the pool contains three essentials daemons which are the *Collector*, *Startd* and *Master*. A pool is a set of computers in which the head node is the Hawkeye Manager and the other computers serve as Hawkeye monitoring agents. The Hawkeye Manager is responsible for collecting all monitoring information which it stores in a database, and for handling user queries. Moreover, the Monitoring Agents send information in the form of ClassAds to the Manager at specified intervals.

The data which *Hawkeye* provides can be accessed using a command interface, a graphical

interface or a web portal. Furthermore, *Hawkeye* allows arbitrary sensors to be installed and it supports the same platforms as Condor.

Future work includes plans for developers to store past monitoring information into a round-robin database which will enable an administrator to study the behaviour of a machine over a given period. Moreover, *Hawkeye* will allow an administrator to detect problems in a resource pool by using a technique called *ClassAd matchmaking*. Subsequently, an administrator will be in a position to execute jobs, depending on various attribute values of pool members.

Critical Analysis of Hawkeye

The disadvantages of *Hawkeye* are its basic front-end interface for the display of statistics and the fact that it is still in early development.

2.5.6 RedLine: A Constraint Language Approach to Resource Selection

Based on the Condor matchmaking concept, *RedLine* [73] is a prototype which extends this idea and regards matching as a generalised constraint satisfaction problem. *RedLine* is more powerful than ClassAds in that it can describe resources with different levels of complexity and generality. For example, ranges may be specified for a resource attribute in *RedLine* whereas *ClassAds* only carries out exact matches on properties. Moreover, as policies form an important criterion for resource selection, *RedLine* utilises both properties and policies to match requirements. Whilst ClassAds performs only one-to-one matches, *RedLine* allows multiway matches which return sets of resources as well as individual resources meeting a particular requirement.

At the core of the *RedLine* system is the handling of matching as a constraint satisfaction problem, and the application of constraint-solving technologies to implement resource search and selection functions. A *Constraint Satisfaction Problem* is formally defined by

a set of variables, each of which has a discrete and finite set of possible values, known as the *domain*. There is also a set of constraints among these variables. A CSP is solved by finding a value for each variable, from their respective domain, which collectively satisfy all the constraints.

A resource request allows the specification of resource requirements and interrelationships; for instance, "two resources with a CPU with a minimum of 800 MHz and a hard disk with at least 2 MB of free space, both located in the same administrative domain". The receipt of a resource request triggers the matchmaking process whereby resources have to be selected from a resource pool, in such a way that all request requirements and resource policies are satisfied. Matching as part of a CSP can be formalised by associating a variable with every requested resource. Moreover, the description of relations which must be present, can be done by placing constraints on variables. This allows the expression of requirements about the resources and their access policies. Subsequently, a constraint-solving algorithm can be used to solve the problem of matching the constraints on the values of these variables.

The requirement for the description of a CSP is a modelling language which allows both requests and resource descriptions to have a declarative representation, and which enables constraints to be placed on the structured data types which describe complex resources. Since existing modelling languages do not support constraints other than those dealing with integer and ground term valued variables, and the need for users to code their own programs to solve problems, the new description language *RedLine* is created.

Furthermore, the *RedLine* system consists of a layered architecture where the *RedLine language* defines the syntax and basic semantics for the specification of *descriptions*. Each description is a collection of constraints which could represent either a request or a resource. Additionally, this system allows a *vocabulary* to be specified so that upon its consultation, constraints can be checked for with more expressive capability. The vocab-

ulary can make use of an ontology language including DAML+OIL [17] and OWL [22], to attach semantics to words in a description. Therefore, *RedLine* can carry out semantic matches by using the semantic information defined in the vocabulary. The layered architecture of the *RedLine* system also consists of a matchmaking engine and a conflict-check engine; top-layer components are composed of the Grid resource selection service, an e-commerce searching engine and other matchmaking services.

The purpose of the *conflict-check engine* is to merge RedLine language statements and the semantic content defined in the vocabulary to verify the consistency of a set of constraints. As far as the *matchmaking engine* is concerned, it implements the logic used to match one request description with multiple resource descriptions. This matchmaking is carried out in two steps: firstly, the resource selection problem is translated into a CSP problem which specifies the required attributes of the request as well as existing resources. Then, the conflict-check engine is executed in order to capture any conflicts arising from the assignment to variables.

The *RedLine* description language has been designed to take into account several aspects including the ability to express requests which refer to resource sets, the flexibility in describing requirements and the criteria used for selecting from multiple matching descriptions, the support for a symmetric description for both resource and resource requests, and the ability to query descriptions based on criteria about their properties and requirements. Furthermore, every description can be interpreted as either a resource advertisement or a request. The difference in the role of the description is determined by the way in which the description is sent to the matchmaker. *Resource* descriptions are submitted through a resource advertising interface while *request* descriptions are submitted through a request advertising interface. All the advertisers use the same terminology so that the description can be understood by all users.

Critical Analysis of RedLine

While *RedLine* can express constraints about resource requests and providers effectively and implements a matchmaking process which uses constraint-solving methods, its use with a wider range of applications has not been evaluated. Moreover, the real-time performance of such a system has not been investigated in terms of the cost of the constraint solver as well as realistic resource pool sizes and structures. Additionally, there has not been any work yet concerning the most appropriate organisation of the descriptions in the matchmaker, or the organisation of distributed information and the preservation of properties. There is also the issue of the complication of the specification as a result of the richness of the interface.

2.5.7 Grid Monitoring Architecture (GMA)

The *Grid Monitoring Architecture (GMA)* [113] is a specification which was proposed by a Grid Monitoring and Performance working group at the Global Grid Forum (GGF) [40]. The GMA addresses the basic architecture for Grid monitoring systems, specifying the required functionalities for each component, as well as the interoperability between different Grid monitoring systems. There are three important components in the GMA specification, as shown in Figure 2.5:

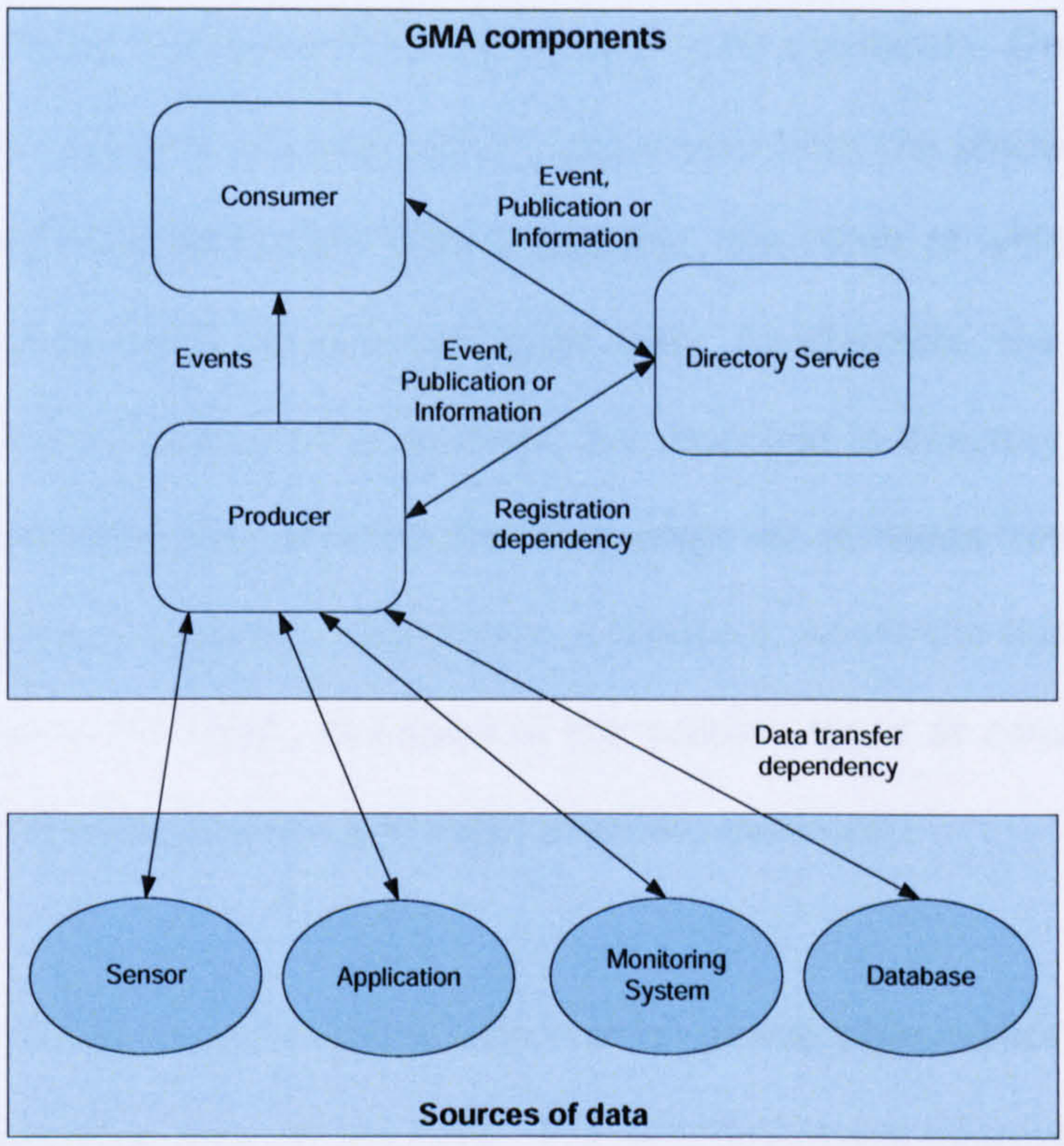

Figure 2.5: GMA Components and Data Sources.

- **Producer** collects performance data from a number of *sensors* and makes it available to other GMA components. Producers can be regarded as *data sources* and they also add or update directory service entries describing events which they will send to a consumer. A producer can also accept query or subscribe requests from a consumer. However, the GMA specification does not specify how the Producer and the sensors mutually interact. Additionally, producers can be used for providing access control to event data, therefore allowing various types of access to different classes of users. Other services offered by producers include event filtering, caching and intermediate processing of the raw data as requested by a consumer. The information regarding the services provided by a producer is published in the directory service with the event information.

- **Consumer** is able to receive performance data from a producer, after which it

processes the data, for instance by filtering or archiving it. Being considered as *data sinks*, consumers first search the directory service for a producer. Once a producer is located, the consumer requests one or more events from the producer. Consumers can also request subscriptions from a producer, the result of which are details of the subscription which are returned in the reply. Furthermore, the events which a consumer will accept from the producer, are described in directory service entries. During *notification*, the consumer accepts a single set of events from a producer. It can also accept a subscribe request from a producer, where the subscription details are returned in the reply. Examples of the possible types of consumers are data archivers, real-time monitors and event overview monitors.

- **Directory Service** is a distributed service which provides the publication and searching of performance data on the Grid. The directory stores information about producers and consumers which accept requests. Along with the publication of their existence in the directory service, producers and consumers also publish the type of events they produce or consume, as well as information about accepted protocols and security mechanisms. Such *registration* information allows other producers and consumers to discover the types of event data which are accepted, the characteristics of such data and methods of access to the data. Nevertheless, the directory service does not store any event data, only publication information about the event instances that can be provided or accepted. Additionally, the directory service can optionally provide the event schema. As well as adding and modifying entries about producers and consumers, the directory service carries out searches on behalf of a producer or a consumer, based on particular selection criteria. Single or multiple matches are allowed and should be specified by the client.

Critical Analysis of the GMA

Since the GMA components and external sensors are distributed and interdependent, it is important that dependencies are kept at a low overhead so that the automation of the monitoring system can be achieved at minimum costs. There are two types of dependencies that may arise amongst GMA components, including its sensors: *data transfer dependency* and *registration dependency*. Both GMA components and external sensors communicate monitored data with one another using networks, resulting in data dependencies. However, difficulties may arise from inaccessible networks or inefficient communication, which result in unreachable sensors. A solution would involve analysing the data dependency information so that consumers can receive relevant events. *Registration dependency* occurs when producers, consumers and sensors register themselves with the directory service, bringing about a central dependency on the directory service. There is an inherent assumption that the existing registrations will be available and accessible continuously from the directory service. However, the directory service might fail or changes in the system might result in the component registration.

R-GMA

Being an implementation of the GMA architecture, *R-GMA* (Relational Grid Monitoring Architecture) [10] was developed as part of Work Package 3 [127] of the EU *DataGrid project* [107] between the years 2001 and 2003. The architecture offers services for providing information, monitoring and logging in a distributed computing environment. Moreover, this architecture combines relational database and Java servlet technologies, where relational databases are used for registration, and producers, consumers and registries are implemented as Java servlets. Producers publish information into R-GMA and consumers subscribe to it. R-GMA collates all the relevant information and presents the view of a single, large relational database which users can query to find the information required. R-GMA also views the information resources in a Virtual Organisation as a

single *virtual database* containing a set of *virtual tables*. Both the Producer and Consumer provide a layer of abstraction around the Registry so that any source or sink for the information does not need to interface directly with the Registry. Additionally, far from being a general distributed RDBMS (Relational Database Management Systems), R-GMA offers a relational model for the Grid environment. However, information producers are relatively independent of one another. Relational attributes are present when Producers announce the content of their publication via an SQL CREATE TABLE statement and publish *tuples* or database rows with an SQL INSERT, and when Consumers use an SQL SELECT to collect the information they need [18]. R-GMA utilises a standard query language which is a subset of SQL. Each virtual table has a *key* column which is declared in the schema. Moreover, each tuple published by a primary producer carries a *time-stamp* so that support is provided to monitoring systems which require time-sequenced data. The registry is responsible for selecting the most appropriate producers to answer a query and this process is called *mediation*.

Subsequently, the consumer contacts each producer directly, combines the information thus retrieved and returns a set of tuples to the user who is unaware of this mediation process. Grid services and applications can interact with R-GMA using several APIs including Java, C, C++ and Python ones. Other methods of access include a web browser and a command line tool which allows users to understand and experiment with producers and consumers.

The R-GMA schema which defines the name and structure of each table type, is either predefined at system installation time or is defined by applications, leading to a very flexible system. Additionally, typical deployments comprise of Producer and Consumer services on a one per site basis, as well as a centralised Registry and Schema. The Registry and Schema may also be replicated to avoid a single point of failure.

Uses of the R-GMA include a Grid resource discovery, monitoring and logging tool. The system is used to discover an appropriate resource when a Job Manager requires such a resource to run a submitted job. On behalf of the user, R-GMA also keeps track of the progress of the job whilst being transparent to the user. R-GMA conforms to the Web Services Architecture [120] where each service consists of a well defined set of *operations* which it can carry out and which are specified in a machine-readable XML document conforming to the Web Services Description Language (WSDL) [121]. One WSDL document accompanies each service with which there is an exchange of messages specifying the operations requested by applications. The WSDL also specifies the format and sequence of messages as well as parameters, needed for each operation. Moreover, R-GMA uses Simple Object Access Protocol (SOAP) [100] messages over HTTPS in a request and response pattern for user-to-service and service-to-service communications.

Upon the successful completion of Work Package 3, the outcome from the DataGrid research and development is now being used as the basis of the Information and Monitoring research activity [59] of the EGEE project [28].

2.5.8 Ganglia

Ganglia [39] is a scalable distributed monitoring system designed for high performance computing systems including clusters and Grids. Based on a hierarchical design aimed at federations of clusters, it relies on a multicast-based listen/announce protocol to monitor state within clusters. It also uses a tree of point-to-point connections amongst representative cluster nodes to create cluster federations and to aggregate cluster state. Moreover, Ganglia leverages widely-used technologies including XML for data representation, eXternal Data Representation (XDR) [96] for compact data transport and RRDTool (Round Robin Database Tool) [97] for data storage and visualisation. Ganglia's careful design of data structures and algorithms enables it to achieve low per-node overheads, high concurrency, robustness and portability to other operating systems and processor architectures.

The system status information gleaned by Ganglia can be made available via a web interface. Furthermore, this monitoring system allows status information to be subscribed to and aggregated across multiple virtual organisations. The advantages of Ganglia are its efficiency in monitoring the status and utilisation of Grid resources and clusters, as well as its optimisation targets on behalf of Grid applications.

The architecture of Ganglia includes three modules, namely *gmond* (Ganglia Monitoring Daemon), *gmetad* (Ganglia Meta Daemon) and the Ganglia *Web portal*. *gmond* is a Java service residing on the client side, which collects data about computer components. That data is saved into an XML document and is subsequently sent to a multicast address. *Gmond* also listens to the state of other Ganglia nodes via a multicast channel. Installed on a chosen head node, the *gmetad* module collects data from a specified group of nodes. This module can also be configured so that it transmits data which it receives to the multicast, and collects data received from other *gmetad* services. This mechanism allows several clusters to be monitored simultaneously. Additionally, the Web portal runs on the same node as the *gmetad* service, and it displays data collected by *gmond* and *gmetad* services in the form of HTML graphs. The historical data presented is saved in round-robin databases. The Web portal therefore allows cluster, hosts and host metrics to be viewed in real-time.

The advantages of Ganglia include its intuitive Web portal interface, its use of the XML data format, its scalability, its use of Java and hence its portability, and its straightforward installation. Furthermore, it allows commands to be executed on the computers it monitors and is a core component of several cluster distributions including Rocks, OSCAR and Warewulf. The monitoring system can also easily add new nodes and it can conveniently integrate with other monitoring systems including the Globus MDS and MonALISA.

Ganglia is used in three types of distributed systems namely clusters, Grids and planetary-scale systems [76]. As each type of system poses a different set of constraints, the design

of Ganglia had to encompass various trade-offs in terms of the physical organisation of these systems and the costs of using different types of resources. In brief, clusters consist of a number of nodes which are connected by high bandwidth and low latency links. Being within a single administrative domain, these nodes are usually homogeneous in both hardware platform and operating system, and are closely coupled. On the other hand, Grids are characterised by sets of heterogeneous resources from different administrative domains, some of which are interconnected by dedicated, high-speed wide-area networks. Planetary-scale systems are wider-scale distributed systems which span a significant proportion of the planet. These systems take the form of overlay networks over the Internet. Moreover, several network drawbacks need to be taken into consideration, notably that high-speed wide-area bandwidth is not as available as clusters due to their high costs. Furthermore, in planetary-scale systems, network congestion and partitions are more commonplace.

Ganglia is currently being used in an open, shared planetary-scale application testbed called *PlanetLab* [91]. PlanetLab runs on over 100 nodes across 42 sites on the North American, European and Australian continents. The original design for Ganglia was to connect clusters with fast local-area networks, but PlanetLab is aimed at providing services on overlay networks. However, overhead issues have been solved and further changes are being made to the Ganglia architecture.

Critical Analysis of Ganglia

The main disadvantage of Ganglia is the load that may be introduced. For example, parsing XML data can be computationally demanding and consequently, this can create network overhead. Moreover, since Ganglia is a Java-based monitoring system, the Java VM (Virtual Machine) could cause extra load on the client side. Additionally, even though the Round-Robin Database Tool (RRDTool) is a very scalable database, it is unclear how data storage will be managed in large volumes and across administrative domains.

Furthermore, since Ganglia can be used in different types of distributed systems, its design has to take into account the various sets of constraints present. For example, Ganglia uses a multicast-based listen/announce protocol for monitoring the state in a single cluster. This can cause high load and bottlenecks on the cluster during service discovery and service notification. Even though Ganglia offers several advantages including the real-time addition and removal of nodes, the automatic configuration of cluster membership and the availability of information about the whole cluster to a single node, it assumes a native multicast service. Nevertheless, this assumption breaks down for distributed systems, including Grids as they rely on wide-area communications. Therefore, Ganglia may not scale efficiently for distributed systems spanning a large geography [98].

2.5.9 NetLogger

NetLogger (Networked Application Logger) [114] is a methodology for analysing and troubleshooting distributed applications. It is designed to monitor the behaviour of various elements of the application-to-application communication path in real-time. This approach helps to discover the location of bottlenecks in a complex distributed system. Bottlenecks are at times the results of interactions between components; they are sometimes also due to unrelated network activity influencing the distributed system. Moreover, when distributed application components use NetLogger, they are adapted to produce time-stamped logs of relevant events occurring at important locations in a distributed system. Subsequently, the events from each component are correlated, allowing the detailed characterisation of the performance of the various aspects of the system and network. This approach has proven to be a very useful tuning and debugging technique for distributed application developers. The management of the large amount of logging data is handled by a Java agent-based system.

NetLogger therefore allows operational problems to be recognised which in turn enable distributed components to adapt to operational conditions. This process aims to reduce

the adverse performance impact on users. Furthermore, NetLogger is designed to be extremely lightweight, and includes a mechanism for reliably collecting monitoring events from multiple distributed locations.

It is the responsibility of software agents to collect and filter event-based performance information, adapt the monitoring of the current distributed system and manage the large amounts of data which is logged. The monitoring adaptation occurs through the fine-tuning of different monitoring options. Moreover, the aim of characterising the performance of distributed elements is to create high-speed components which can be used for building high-performance applications, instead of fine-tuning the applications themselves in a top-to-bottom fashion. This approach can also provide information to applications adapting to component congestion problems.

The components which can be instrumented to produce monitoring include application software, middleware, operating systems and networks. All monitoring events are also required to use a common format and set of attributes, and be synchronised globally. Therefore, the clocks of all hosts participating in the distributed system should be synchronised. Furthermore, NetLogger logs all disk and network I/O, as well as the start and end of a program execution or software component. The event logs produced by NetLogger, include high-resolution, synchronised timestamps which are recorded both before and after the events. Furthermore, the activities about which events are logged, include application, operating system and network ones.

The Netlogger Toolkit consists of four components which are:

- A library of functions and client APIs to generate application-level monitoring event logs. Calls to these APIs from the existing application source code, can be implemented in C/C++, Java, Perl or Python. The library is designed to be as lightweight as possible and to never block or adversely affect application performance. An environment variable is also available to vary the destination and logging level of

NetLogger messages.

- A set of host and network monitoring tools which can interoperate with other monitoring systems including Ganglia and MonALisa [85],

- A set of tools for collecting and managing log files. A daemon called *netlogd* collects NetLogger events from multiple points at a single, central host. There is also an event archive system for NetLogger data, which utilises the mySQL database [111]. Another tool allows the forwarding of NetLogger files in a specified directory to a specific location.

- A visualisation and analysis tool for the log files, called *nlv* (NetLogger Visualization tool). This customisable, X-Windows tool is useful for viewing and analysing event logs based on time-correlated or object-correlated events. *nlv* also allows the browsing of historical data and the playback of specific time periods. Therefore, system-level and application-level events can be represented graphically in a flexible and interactive manner.

An application developer inserts calls to the NetLogger API at important points in the code, which links the application with the NetLogger library. This step subsequently allows the application to be instrumented for the production of event logs. Moreover, the approach which the NetLogger analysis method uses is the generation of *lifelines* representing the workflow of a distributed process. Lifeline analysis enables an intuitive investigation into where time is most spent, using the slope of the line.

NetLogger has proved to be a very useful tuning and debugging methodology for distributed application developers, by being able to correlate detailed application instrumentation data with host and network monitoring data, based on the timestamps. The level of monitoring offered, helps the identification of bottlenecks, performance optimisation and research into network performance. By combining network, host and application-level monitoring, NetLogger is able to provide a complete view of the entire system. Moreover,

the NetLogger methodology can be adapted to any distributed system architecture, even though its past application includes a loosely-coupled client-server architecture. The additional benefit is NetLogger's independent behaviour. Additionally, NetLogger is a valuable tool for debugging multi-threaded programs and it allows the visualisation of interactions between threaded components in terms of the timing of their execution and whether they are blocking.

Some of the later changes to NetLogger include a highly efficient binary format which reduces NetLogger overhead, a reliability mechanism which diverts NetLogger messages to a second location if the primary one is unavailable, and an activation mechanism which allows the level of monitoring of a running process to start, stop or be modified.

The relatively new NetLogger `Reliability` API contributes fault-tolerant features to the distributed system. The solution implemented is based on a temporary fail-over destination for the monitoring data. During a failure, data will be transferred to a destination specified as the result of an API call. Subsequently, the library checks that the original destination has recovered and if that is the case, the data which has been logged during the failure, is sent over.

The NetLogger API also provides a *trigger* function which makes the library verify for changes in the log destination at user-specified intervals. The two types of triggers offered are a *file trigger* and an *activation trigger*. The file trigger scans a configuration file and the activation trigger enables users to activate different levels of NetLogger instrumentation via activation messages destined for an activation service. These triggers allow NetLogger's behaviour to be changed dynamically inside a running application.

Another component of the NetLogger toolkit is the *Monitoring Activation Service* which is used to control the start and end of application-level monitoring in an instrumented application. Applications are required to use the NetLogger *trigger* API which uses an external file-based mechanism similar to the configuration files used by *log4j*. The moni-

toring activation service is continuously waiting for requests to send data to a consumer. On receiving a request, the service creates a trigger file entry for the given event type, and sends NetLogger output to itself. The output is then read and buffered to disk. It simultaneously reads data from the disk buffer and sends it to the original consumer. Moreover, it is possible for the activation service to apply a client-specified filter to the monitoring data stream before it is buffered and forwarded.

Critical Analysis of NetLogger

The distributed environment in which NetLogger has been used is in loosely-coupled, client-server architectures. Even though this was the principal usage mode, NetLogger can theoretically be adapted to any distributed system architecture. However, their integration depends on the design of the other system, and it is unclear how straightforward this integration is.

There are several constraints brought about the NetLogger toolkit in terms of overhead to other systems. For example, NetLogger monitoring should be limited to events which are longer than a few milliseconds, considering that a call to the NetLogger client library to generate an event log, takes between 0.2 and 0.5 milliseconds. Furthermore, NetLogger should have the minimum intrusion possible on the system being monitored. Care must also be taken to prevent NetLogger messages from causing excessive network traffic, and subsequently it should be possible to enable or disable logging.

Additionally, the message formats supported by NetLogger include ASCII and the binary message format. The latter format is quicker than ASCII but is more difficult for third-party tools to utilise. Moreover, NetLogger provides tools for converting between the ASCII and binary formats. Previous versions of NetLogger used the IETF-proposed (Internet Engineering Task Force) Universal Logger Message (ULM) format which was easy to read and parse, but resulted in unnecessary overhead. Consequently, the NetLogger team

developed the new binary format which utilises the same API with the advantages of being smaller and quicker.

2.5.10 MonALISA

MonALISA (MONitoring Agents using a Large Integrated Services Architecture) [82, 85] is a distributed service for the monitoring, control and global optimisation of complex systems. The scalable system is based on a collection of autonomous multi-threaded, agent-based subsystems which are registered as loosely-coupled, self-describing dynamic services. These subsystems also collaborate in performing a wide range of monitoring and decision-making tasks on behalf of large-scale distributed applications, and allow the information gathered to be discovered by other services and shared with clients. Another feature of MonALISA is the organisation of its services into groups, which is used for registration and discovery. The monitoring information thus gathered allows components to provide decision support and subsequently, automated decisions that help to maintain and optimise workflow throughout the Grid.

Moreover, the MonALISA system is developed using a scalable Dynamic Distributed Services Architecture (DDSA) which utilises technologies including Jini [4, 58] and Web Services. The functionalities supported by MonALISA include a dynamic service system and the ability to be discovered for the provision of relevant, filtered information. Its approach is to provide monitoring information efficiently from various distributed locations to a set of loosely coupled higher-level services. Moreover, the framework allows existing monitoring tools and procedures to be integrated in the system for aggregating information about computational nodes, applications and network performance.

Being able to discover dynamically all the *farm units* within an organisational unit, MonALISA also supports remote event notification for changes in the system. Global monitoring repositories are therefore maintained for various virtual organisations. MonALISA also

provides a secure mechanism for the dynamic configuration of monitored elements and the collected information. Furthermore, subscribed listeners can access selected real time data. Historical data is also made available via a persistent mechanism based on JDBC (Java Database Connectivity). Additionally, the framework consists of mobile agents which are responsible for controlling different activities in the system. Other components of the framework include configurable GUIs that aggregate real-time information from multiple farms into a single view.

In the DDSA framework, a service interacts autonomously with other services via dynamic proxies or agents using self-describing protocols. Transparent communication amongst services is possible through dedicated lookup services, a distributed services registry, and discovery and notification mechanisms. The lookup discovery service automatically notifies subscribed services when new services become available, even when a notification provider was not present at registration time.

Mobile agents and dynamic proxies make use of the code mobility paradigm which extends the methodology in remote procedure call and client-server. The dynamic download of both the code and the relevant parameters into the system has several advantages including optimised asynchronous communication and disconnected operation, remote interaction and adaptability, dynamic parallel execution and autonomous mobility. MonALISA is also able to build an extensible hierarchy of services which is capable of managing very large systems, by combining service architecture with code mobility.

Moreover, each MonALISA service registers with a set of JINI *Lookup Discovery Services* which are part of a group possessing specific attributes. Each of the JINI Lookup Discovery Services can register itself to other services, so that information belonging to common groups is replicated in those groups whenever a change of state is detected. MonALISA can therefore build a distributed and reliable network for the registration of services. MonALISA users can select services, based on a set of matching attributes.

To access real-time or historical data, clients can use a predicate mechanism for the request or subscription to selected measured values. Built on regular expressions, these predicates are used to match the attribute description of the measured values in which a client is interested. Furthermore, the MonALISA architecture provides a proxy service which clients use to connect to various services. Based on a JINI service, the proxy service mutually discovers other services, especially when the latter run behind a firewall. Subsequently, clients can interact with these services via the proxy services. The latter are also used for redundancy and for dynamically load-balancing clients.

Critical Analysis of MonALISA

Being multi-threaded, MonALISA is able to keep the load on host systems to a minimum. This is done by creating a dynamic pool of threads only once and by reusing threads when tasks assigned to those threads are completed. This enables a large number of monitoring modules to run concurrently and independently and also for the load to be dynamically adapted. Other advantages include the continuation in execution of monitoring tasks while others hang or fail. MonALISA also provides monitoring modules for pulling data, which can be run frequently and can be dynamically loaded from several predefined centralised sites. However, users would need to know beforehand the location of these sites and the way of accessing them. Moreover, sites could be down or users would not be able to access sites when network links are severed.

2.5.11 DiPerf

DiPerf [24] is a performance-testing framework which has been developed to facilitate and automate the performance evaluation of services. The main purpose of this framework is to discover the scalability limits of a service, which is the maximum offered load, whilst an acceptable quality of service is still being provided. Moreover, LAN-based tests are not

deemed sufficient for measuring the service performance experienced by heterogeneous, geographically distributed clients with various levels of connectivity. This is the problem which DiPerf attempts to solve by providing accurate estimates of the service performance experienced by distributed clients. In brief, DiPerf had been developed to act as a practical tool for the automated evaluation of service performance, and for future evaluation of performance on behalf of service developers.

DiPerf monitors a distributed pool of machines which are running a target service and it collects performance metrics which are analysed and interpreted. The data collected imparts knowledge about the service's maximum throughput, service *fairness* when many clients access the service concurrently, and the influence of network latency on service performance. Performance models can consequently be developed for finding the correlation between service performance and offered load. The estimates produced by these models can then be used by resource schedulers, which can maximise resource utilisation but can still offer an adequate quality of service. Moreover, DiPerf automates the deployment of a service and its clients, as well as collects, tests and analyses data.

The DiPerf framework is made up of two components: the *controller* and the *testers*. The controller is responsible for distributing the service code which it receives from clients, to testers. Furthermore, the controller coordinates the execution of the testers and aggregates their performance measurements. While running the code on its machine, each tester times the network calls which the code makes to the target service.

The DiPerf framework includes a set of candidate nodes from which testers are chosen. Moreover, future work involves the extension of the framework for allowing the selection of a subset of available tester nodes to meet requirements including link bandwidth, latency, compute power, available memory or processor load. Additionally, metrics are aggregated either at the testers or at the controller. It is also possible for additional metrics to be reported to testers, for example those collected by clients, via supplementary interfaces.

These metrics are then forwarded on to the controller for statistical analysis. Moreover, DiPerf performs service evaluation in real-time using performance data which testers send to the controller.

For reliable results to be aggregated at the controller, time synchronisation is of the essence. Consequently, DiPerf needs to ensure that all its clients' times are synchronised. This is done by using a timer component which enables nodes to query for a global time and to allow the time on these nodes to synchronise within tens of milliseconds' accuracy. However, since synchronisation is not performed online, DiPerf calculates the offset between local and global times. DiPerf then applies that offset to aggregated metrics when analysing them.

Individual testers gather service response times which are aggregated by the controller, and correlated with the offered load, and with the start and end times for each tester. Subsequently, the controller is able to deduce the service throughput and the ratio of the number of jobs completed and service utilisation. DiPerf describes *service utilisation* as the ratio between the number of requests handled for a client and the total number of requests handled by the service during the time the client was active. As all the metrics gathered share a global timestamp, DiPerf can easily combine all the metrics in properly defined time denominations to produce an aggregate view of the service performance.

Critical Analysis of DiPerf

At the time of writing, DiPerf developers plan to perform similar experiments as with GT3.2, for GT4.0 pre-WS GRAM. It is not known at this time whether performance could be significantly improved with the GT4.0 usage of lightweight WS-Resources, compared to GT3.2 Web Services GRAM. Additionally, it is unclear how accurate the empirical models developed using DiPerf are, and the type of applications which could potentially benefit from them in terms of improved resource allocation decisions. Moreover, experiments

have not been performed to date to verify the claim that the DiPerf framework could easily scale up to thousands of nodes.

2.6 Event Services

In this section, examples of systems with event models are described briefly.

2.6.1 ECho

ECho [27] is a high-performance event-delivery middleware which provides reliable binary transmission of event data. The event data possesses distinguishable features which support data-type discovery and organisation-wide application evolution. This middleware meets the demands of the Grid environment, whereby components are assembled ad hoc, from distributed locations to execute an application. Moreover, the communication supported by ECho is the *publish/subscribe* model. It is therefore possible to introduce new components to an ECho-based system by their registering to an appropriate set of events, without any need for recompilation or re-linking.

The main advantage offered by *ECho* is its efficiency in transmitting events across heterogeneous machines. This is achieved by ECho recognising user-defined event formats and by providing runtime translation for those. ECho also provides *type extension* where it transparently extends existing data types without impacting existing code which uses the old type. Component-based systems also rely on *reflection* which allows third parties to discover the contents of a data type on the fly and to operate on the latter without having any a priori knowledge. Moreover, the major feature of ECho is the provision of the above characteristics whilst still maintaining a high level of performance.

The events handled by ECho are similar to those in event delivery systems that use channel-based subscriptions. It is through *channels* that the extent of event propagation is controlled, and hence the matching of event sources and sinks. Additionally, the event

channels in ECho are lightweight, distributed entities which are organised in the same virtual space.

Not only does ECho provide interprocess event delivery, but it also enables threads to be associated with event handlers. This feature allows intra-process communication, as well as the transparent handling of both inter- and intra-process communication by the event sender. Subsequently, local and remote sinks may both appear simultaneously on a channel.

2.6.2 Common Object Request Broker Architecture (CORBA)

The *Common Object Request Broker Architecture (CORBA)* [29] is an open distributed object computing infrastructure which is standardised by the Object Management Group (OMG). CORBA automates several network programming tasks including object registration, location and activation, by providing a set of distributed services to support the integration and interoperation of distributed objects. A standard CORBA [27] request results in the synchronous execution of an operation by an object. When that operation defines parameters or return values, data is communicated between the client and the server. Moreover, as a request is directed to a particular object, both the client and the server need to be available.

The *CORBA Event Service* decouples the communication between objects. *Suppliers* produce event data and *consumers* process that data. Furthermore, CORBA provides two models for initiating event communication: the *push* and *pull* models. The push model enables suppliers of events to initiate the transfer of event data to consumers. In the pull model, the consumer starts the transfer of event data with a request.

The main component of CORBA is the Object Request Broker (ORB) which encompasses all communication infrastructure required to identify and locate objects, handle connection

management and deliver data. The ORB passes requests from clients to the object implementations on which they are invoked.

Additionally, an object provides services via its *interfaces* defined in OMG's Interface Definition Language (IDL). These object references also allow distributed objects to be identified. Moreover, the emphasis placed by the CORBA specification is that clients and object implementations are portable. The specification defines APIs for clients of distributed objects, as well as for the implementation of the objects themselves. Subsequently, one vendor's CORBA product can be adapted to work with another vendor's product, for both client and object implementations.

2.6.3 JINI

Originally developed by Sun Microsystems, *Jini* technology [4, 58] is an open software architecture which enables dynamic networking in Java for building adaptable distributed systems. This technology is used to create systems which are scalable, evolvable and flexible, and that are usually required in dynamic runtime environments. Jini services can represent hardware, software or both, and a collection of such services forms a Jini federation. The main objective of Jini is to convert local- and wide-area networks into a flexible, efficiently managed system which human and computational clients can use to discover services effectively and robustly.

The Jini technology is designed to incorporate change in distributed systems. One of the ways in which Jini creates dynamic environments is by building systems that implement a service-oriented architecture. Discrete services provide functionality on the network, and are represented by objects called Jini service *proxies*. A client subsequently invokes methods on a service proxy which fulfills the promised service. Java-based service proxies also enable clients to interface uniformly to both hardware and software services, without

any need for specific resource knowledge.

When a client searches for a service on the network it starts by sending out a query by multicast to discover a *Jini Lookup Service*. After the latter has been discovered, the corresponding remote object is copied to the client's machine. This object is consequently used to access the required service. Furthermore, service discovery is carried out by interface matching or Java attribute matching. On the other hand, if the Jini Lookup Service contains a valid service implementing the interface specified by the user, a proxy for that service is downloaded into that user's machine. Subsequently, the proxy is used to call the other functions provided by the service, without any further involvement from the lookup service.

Additionally, each service has to discover one or more Jini Lookup Service before being allowed to join a federation. The location of the Jini Lookup Service may be known beforehand or it can be discovered using multicast. Moreover, group names can be assigned to a Jini Lookup Service so that another service can discover it in close proximity.

Changes in a distributed system occur when new resources join the network or when existing ones leave. Jini technology allows clients to discover services which have recently arrived or been relocated on the network, by providing a *lookup service*. Jini also provides a set of *discovery protocols* which clients use to find lookup services whose location is not known a priori. For example, a client can use the Jini lookup service if it wishes to utilise that service for the first time.

Additionally, in the Jini architecture, service functionalities and capabilities are described in Java object interface types. Matching of service capability occurs at the object and syntax levels, where exact semantic matching occurs for discovering services. Therefore, inexact matching is unavailable in Jini.

2.7 Semantic Web

The *Semantic Web* [63] which is built on the current web technology, allows information to be given well-defined meaning. This technology can be used to describe entities including data, services and resources, and it allows resources to be queried and matched, based on their capability rather than on their syntactical expression. In brief, the goal of the Semantic Web is to develop enabling standards and technologies for increased semantic expression allowing support for richer discovery, data integration, navigation and the automation of tasks. The use of the Semantic Web results in more accurate search results, tighter integration of information from different sources and the comparison of relevant information.

A wide range of identifiers are used to refer to various entities. Moreover, resources on the Web consist of Web documents having informative relationships, known as *links*, with one another, for example, `depends on`, `is a version of`, `has subject` and `authors`. The resources and links can also have types associated with them, which define concepts which give more information. For instance, some links may indicate that a resource is a version of another resource or is written by a resource which describes a person. Additionally, the Semantic Web utilises descriptive conventions which can increase with human understanding. These conventions enable the effective merging of the independent, parallel work from diverse communities, although different vocabularies have been used.

The implementation of the Semantic Web principles occurs in the layers of Web technologies and standards. The Unicode and URI layers ensure the use of international character sets and allow objects to be identified in the Semantic Web. Furthermore, the XML layer with namespace and schema definitions ascertain the integration of the Semantic Web definitions with the other XML-based standards. With the Resource Description Framework (RDF) and its schema, statements can be constructed about objects with URIs and vocabularies defined. In this layer, types are attributed to resources and links. The *On-*

tology layer supports the evolution of vocabularies, whilst defining relations between the different concepts. Additionally, the top-level layers: *Logic*, *Proof* and *Trust* are the subject of current research. The Logic later allows the writing of rules while the Proof layer executes those rules. In conjunction with the Trust layer, the Proof layer also evaluates the degree to which the given proof should be trusted.

2.7.1 Universal Description, Discovery and Integration (UDDI)

The *UDDI* (Universal Description, Discovery and Integration) [87, 115] specification defines open, platform-independent standards which allow different organisations to share information via a global business registry, discover services offered in the registry and define how these services interact over networks. UDDI is a key member of the group of interrelated standards making up the Web services stack. A standard method is therefore defined for the publication and discovery of the network-based components of a service-oriented architecture (SOA). The type of data which would typically be provided in a UDDI implementation are *white pages* of business contact information, *yellow pages* which categorise businesses by standard taxonomies and *green pages* which provide technical information about services that are exposed.

Being a major component of the service-oriented approach to software design, the UDDI registry model adds considerable business value to organisations by enabling policy-based distribution and management of enterprise Web services. Moreover, UDDI increases software flexibility, reuse and control through the simultaneous satisfaction of requirements for enterprise architects, developers and the underlying business policies. In short, UDDI is a standard which specifies protocols for the location of a software service by accessing a registry for Web services, invoking that service and managing metadata about that service.

Furthermore, UDDI (Universal Description, Discovery and Integration) registries provide

binding information dynamically at run-time, about the APIs of external services. They can also be made to respond in different ways according to various taxonomies, and other requirements including security, transport and quality of service. Additionally, UDDI is useful in offering an interoperable, standards-based approach for the systematic documentation and publishing of Web services, thereby allowing better code reuse and developer productivity. Moreover, UDDI registries provide a layer of indirection for service-oriented application development and management. Therefore, this layering between a service and the applications which call it, allows for easier changes in the life cycle of specific components, including version updates.

The UDDI specification defines services that support the description and discovery of organisations and other Web service providers, the Web services these organisations make available, and the programmatic interfaces which are offered to access and manage those services. Additionally, UDDI is based on a number of industry standards including HyperText Transfer Protocol (HTTP), XML, XML Schema Definition (XSD), SOAP and WSDL.

For users and applications to be able to use a particular Web service, they require information which is defined by the UDDI XSDs. These core types of information form a base information model and an interaction framework of UDDI registries. Such information includes a description of the service's business function, information about the organisation publishing the service, the service's technical details and several other attributes including taxonomy and digital signatures. Furthermore, UDDI defines a consistent way for publishers to add new classification schemes to their registrations.

Improvements in the UDDI specification led to the current version which uses an open, standardised approach to enable widely interoperable communication. Comparatively, previous definitions of the standard depended on proprietary ways of interaction. Moreover, the UDDI specification also defines the concept of *registry affiliation* whereby infrastruc-

ture topologies are supported: hierarchical, peer-based and delegated. Defining the various links amongst different UDDI registries, these topologies are mapped on the relationships of the underlying real-life business processes.

Critical Analysis of UDDI

UDDI implementations are usually used by businesses to support their own Web services infrastructure. However, possible extensions to include other companies' Web services, are not clear. There is a tendency for existing, current web service applications to be used within organisations or amongst trusted business partners. With time, the UDDI specification has evolved to support different implementations of the standard, including public registries such as the Universal Business Registry (UBR) and private registries which can be deployed on an organisation's internal networks. The UBR is the main directory for the publication of publicly available e-commerce services and is also a public instance UDDI, currently operated by four companies namely, IBM, Microsoft, NTT Com and SAP. However, the number of Web services listed in UBR databases is relatively small, leading to the observation that UDDI may not be as popular for Web services discovery purposes as it was designed for.

Furthermore, there is evidence of political discussions where UDDI founders are torn between keeping their leadership positions and promoting the distributed architecture of the implementation systems. Consequently, conflicts of interest may inhibit the advancement of the standardisation of the technology.

Currently, Internet-based UDDI servers are not being used to discover WSDL interfaces of sought-after Web services. Instead, the WSDL interfaces of the desired Web services are sent internally from one person to another. The UDDI was designed for a distributed environment where applications are required to search for and access widely available Web services, all with frequently changing connections. Nevertheless, a number of issues

face dynamically discovered Web services, including security, market demand and charging mechanisms. Subsequently, the current trend is unfortunately that UDDI implementations are being utilised to discover Web services found within the same administrative domain. This usage does not make good use of UDDI as data within intranet-based projects do not change very often.

There is also the requirement that service providers keep their published data up-to-date for dynamic Web service consumers. Otherwise, the UDDI cannot fully serve its purpose. An API is provided by UDDI for the posting of service information.

2.7.2 Web Service Inspection Language (WSIL)

The *Web Service Inspection Language* (WSIL) [9] is an XML document format which facilitates the discovery and aggregation of Web service descriptions in an extensible manner. The main purpose of WSIL is to complement UDDI as a model for service discovery. It was created by a group of IBM and Microsoft engineers and was released in November 2001. Moreover, WSIL adopts a document-based approach that can leverage existing Web architectures more efficiently and is lightweight. WSIL uses the decentralised model where service description information can be distributed to any remote location using an extensible XML document format. The assumptions with WSIL are that the service provider should already be known and it relies on other service description mechanisms including WSDL. Moreover, WSIL can be extended to support additional information sets which are required by certain service descriptions and aggregations for the process of service discovery.

WSIL represents a specific data entity, the services available, as well as information to access them. Nevertheless, the services which users access, are implemented directly by the providers; WSIL merely advertises their availability. Furthermore, the use of XML as a document format allows innovative applications to be developed and accessed relatively

easily due to the simplicity of XML's functionality. Moreover, WSIL was designed to enable information to be easily authored, published and maintained. In brief, WSIL represents a file format which has references to published Web services, for the aim of discovery and accessibility. Not only does WSIL provide references to service descriptions, it also supports links to other aggregations of service pointers, including other WSIL files or UDDI repositories.

There are a couple of conventions which the WSIL specification uses to allow the different decentralised inspection documents to be discovered. The first convention uses a fixed filename, `inspection.wsil`, that can be located in common entry points. Users are only required to access a uniform URL to verify that the inspection file can be retrieved. The second convention resorts to the use of embedded references in other documents, including HTML or other WSIL documents. This is carried out by using a `meta` tag within an HTML document, which links to the location of the inspection documentation. Some implementations use both conventions, without incurring any extra costs; therefore, service consumers can choose the method they prefer.

Moreover, the extensibility of the WSIL specification is enabled via the use of XML namespaces. The advantage of this extensible mechanism is the evolution of service descriptions and repositories, without the need to revise the base specification.

Critical Analysis of WSIL

The requirement with WSIL is that the service provider should already be known. It also relies on other service description mechanisms such as WSDL. Additionally, two features of WSIL are its low functionality and lightweight nature, which lead to the burden of the implementation to the developer. However, if a document becomes too large in size or the level of nesting in a collection of documents deepens beyond a certain level, WSIL starts to perform inefficiently for searching and document management.

Additionally, the data tags used in the WSIL specification might be rather restrictive, preventing more granular forms of meta-information from particular problem domains from being embedded.

WSIL is an implementation of the WS-Inspection specification. While bindings provide information about referenced documents, there is no guarantee that the ultimate information contained within a WS-Inspection document is accurate. Subsequently, consumers cannot immediately associate the information provided with the actual contents of the referenced document. However, authors of WS-Inspection documents are required to ensure that the information in these documents is as accurate as possible.

Furthermore, WS-Inspection document processors are required to ensure that references are not followed circularly since there are no rules for the way in which link elements are processed.

Being the underlying specification of the Web Services Inspection Language, WS-Inspection is described below.

WS-Inspection Specification

The *WS-Inspection* specification [102] provides an XML format which can help in the inspection of a site for available services, as well as a set of rules describing the usage of inspection-related information. The WS-Inspection document allows references to pre-existing service description documents, to be aggregated; these service description documents are in a number of formats. Consequently, these inspection documents are made available where the service is located, and references can be placed in a content medium such as HTML.

The WS-Inspection specification therefore defines an XML grammar which facilitates the aggregation of references to different types of service description documents, as well as a well defined pattern of usage for instances of this grammar. Accordingly, the WS-

Inspection specification allows sites to be inspected for services that are being offered. It also offers mechanisms which enable existing repositories to be referenced and used; these repositories contain descriptions of Web services. The inherent advantage is the avoidance of information duplication.

The specification is designed to be simple and extensible. The users of the WS-Inspection document are able to choose the service descriptions which they can interpret, at the time of accessing the document. Moreover, when other description formats are created, new references can be added to an existing WS-Inspection document without the base WS-Inspection schema requiring any changes. Furthermore, the WS-Inspection specification enables relevant, retrieval-related description information to be associated with an abstract WS-Inspection entity. Consequently, required documents can be retrieved more easily.

Crucial information is contained within the inspection document in *service description* elements which provide pointers to other documents in various formats. Consumers can subsequently process only the documents that are useful to them. Link elements can also be processed in any pattern; therefore, it is possible for consumers to create different structures from the references in the WS-Inspection documents. Moreover, the WS-Inspection specification can reference WSDL documents with various contents.

2.8 Monitoring Tool Comparison

The distributed monitoring tools described in the previous sections are compared, according to several classifications. This summary is shown in Figure 2.6, where a tick mark is shown in terms of the type of system, the organisational principles, the scalability of the system and the query type.

Monitoring tool/system	System Type		Organisation		Scalability		Query Type	
	Resource-oriented	Service-oriented	Hierarchical	Peer-to-peer	Centralised analysis	Distributed analysis	Query	Subscribe
Globus MDS	✓	✓	✓		✓		✓	✓
GrADS	✓			✓	✓		✓	
NWS	✓	✓		✓	✓		✓	
Condor	✓		✓		✓		✓	
Hawkeye	✓		✓		✓		✓	
RedLine	✓		✓		✓		✓	
GMA	✓	✓	✓	✓	✓	✓	✓	✓
R-GMA	✓	✓	✓	✓	✓	✓	✓	✓
Ganglia	✓		✓	✓	✓		✓	
NetLogger		✓			✓		✓	
MonALISA		✓		✓		✓	✓	✓
DiPerf	✓	✓	✓		✓		✓	
Event Services		✓		✓		✓	✓	✓
UDDI		✓	✓		✓		✓	
WSIL		✓		✓		✓	✓	
WS-Inspection		✓		✓		✓	✓	

Figure 2.6: Characteristics of the various monitoring tools described.

2.9 Summary

This chapter described the architecture of the Grid and its various components. The chronological evolution of the Grid was also detailed, introducing the open source Globus Toolkit as the de facto middleware for Grid computing. Moreover, the various versions of the Globus Toolkit were highlighted as standards and protocols were formalised. An overview of the components of the Globus Toolkit was also presented, including the Grid Information Service. Additionally, an overview and critical analysis of a number of Grid resource discovery and monitoring systems were given. The work described in this thesis also provides valuable support to these systems in the form of administrative domain-independent resource information which is reliable and up-to-date. Moreover, the work in this thesis is based on the Globus Toolkit MDS, of which versions two and three are presented in the next chapter.

Chapter 3

Monitoring and Discovering Resources with the Globus Toolkit®

3.1 Introduction

Users normally expect services from Grid systems and applications to be continuously available and to be of a high quality. Nevertheless, in production systems, it is usual for certain resources to be pulled off from the virtual system at some point, or indeed added to it on a dynamic basis. However, it is a requirement of the Grid that such changes in the availability of infrastructural resources be as seamless as possible, and that the system as a whole should remain available as much as possible.

For this model of availability to exist, it is necessary to both *monitor* and *discover* resources and services. Over the last years, the different versions of the Globus Toolkit have provided various implementations for the subscription to, location and publication of service information. Simultaneously, the Grid community has developed a number of specialised system monitoring tools which can either be used in a stand-alone way or as add-on elements to the Globus middleware.

Two different categories of monitoring and discovery mechanisms are available to Grid applications: **basic** ones and **specialised** components. Specialised system monitoring tools have been developed by members of the Grid community, which can be utilised in a

stand-alone manner or integrated with the Globus architecture. Examples include Ganglia, MonALISA and Hawkeye, which are some of the Grid monitoring software that have been described in Chapter 2. This chapter first overviews the required characteristics of Grid discovery and monitoring systems. It then describes the OGSA-compliant MDS3, followed by an overview of the components and interactions of the LDAP-based MDS2.

3.1.1 Basic Monitoring and Discovery Middleware

The Globus Toolkit and the OGSA architecture from one of its versions, GT3, offer respectively a core architecture and an implementation for locating, publishing and subscribing to resource information. The Globus Toolkit provides Java, C/C++ and Python implementations of the OGSI and WSRF/WSN (Web Services Resource Framework/Web Services Notification) specifications, which are each called *WS Core*. Each WS Core implementation allows web services to be created, offering OGSA features as well as other capabilities. Moreover, both the *service data* feature of OGSI, and the *resource* properties and *notification* features of WSRF/WSN provide a basis for monitoring and discovering Grid services.

Additionally, a WS Core implementation provides a uniform interface for accessing status and configuration information from Web services. After service developers decide on the information which should be available from their services, access to it is provided via the WS Core interfaces. Furthermore, WS Core provides a subscription and notification service which enables clients to subscribe to selected information and to be subsequently notified when the associated information is modified. In brief, the WS Core features including uniform enquiry, and subscription and notification form the basis for both system and application monitoring.

Another component of the Globus Toolkit providing basic monitoring and discovery capabilities is the *Index Service*. It is an OGSA-compliant, aggregator Web service which

collects status information about other OGSA Web services. The Index Service also creates and manages dynamic service data through *Service Data Providers.* These providers offer direct, detailed information about the physical system in a Grid.

At the same time, the Index Service aggregates status information from other Grid services, thus providing a central repository for information from multiple services, to clients wishing to discover dynamic Grid services.

3.1.2 Features of Grid Monitoring and Discovery Systems

GridAdapt (Self-adaptive Grid Resource Monitoring) is a system which we have developed to answer autonomous needs for the discovery and monitoring of Grid resources. The delivery of quality-of-service characteristics (for example, the average query response time, the number of queries serviced per unit time, and the load average) in the efficient management of distributed Grid resources relies on the ability for different nodes to collaborate and exchange information. In this manner, the state of the resources can be discovered and can be acted upon. GridAdapt leverages existing, open-source Grid middleware to achieve its aims. Therefore, the work proposed can be incorporated into projects using such Grid middleware with the added value of reliability.

The nature of the Grid environment [37, 32] can be summarised as a combination of both dynamic, disparate behaviours and intricate geographical compositions. Subsequently, several functions need to be in place for the objectives of the Grid to be realised, including the discovery, monitoring, aggregation and categorisation of participating, shared resources. In order to abstract the complexity of the Grid away from the end users, information services are developed to provide the fore-mentioned capabilities. A *resource* can be any hardware, software or data component which is useful to other Grid entities. Grid information services are required to fulfill the following functions:

- Fast access to data, or a low average response time,

- Uniform and flexible access to static and dynamic resource information,
- Accurate reflection of the status and availability of heterogeneous resources,
- Scalable and reliable access to data about these resources,
- Efficient, cost-effective access to secure data,
- Concurrent access to a large number of geographically dispersed information sources and
- No centralised control of operations.

For the rest of this chapter, the monitoring and discovery components of the Globus Toolkit which form the basis of the research contribution of this thesis are described. Firstly, an overview of the OGSA-compliant MDS3 is given, followed by an explanation of the previous LDAP-based MDS2.

3.2 The Monitoring and Discovery System - MDS3

The Monitoring and Discovery System (MDS3) [51] is the information services component of the OGSA-based Globus Toolkit GT3, and it forms an integral part of the basic services in GT3, as can be seen in Figure 3.1. It provides essential services for the collection, aggregation, subscription, notification and querying of information concerning the status and availability of remote, distributed, heterogeneous resources. The MDS3 architecture also supports both the queries of service data elements and the monitoring of data streams.

The Globus Toolkit 3 (GT3) is built upon concepts defined by the Open Grid Service Architecture (OGSA) [35] where technologies are drawn from both the Grid and Web services communities. This architecture defines uniform service semantics which are exposed as the *Grid service*. It also defines standard mechanisms for the creation, naming and discovery of both transient and persistent services. Moreover, a single Grid service

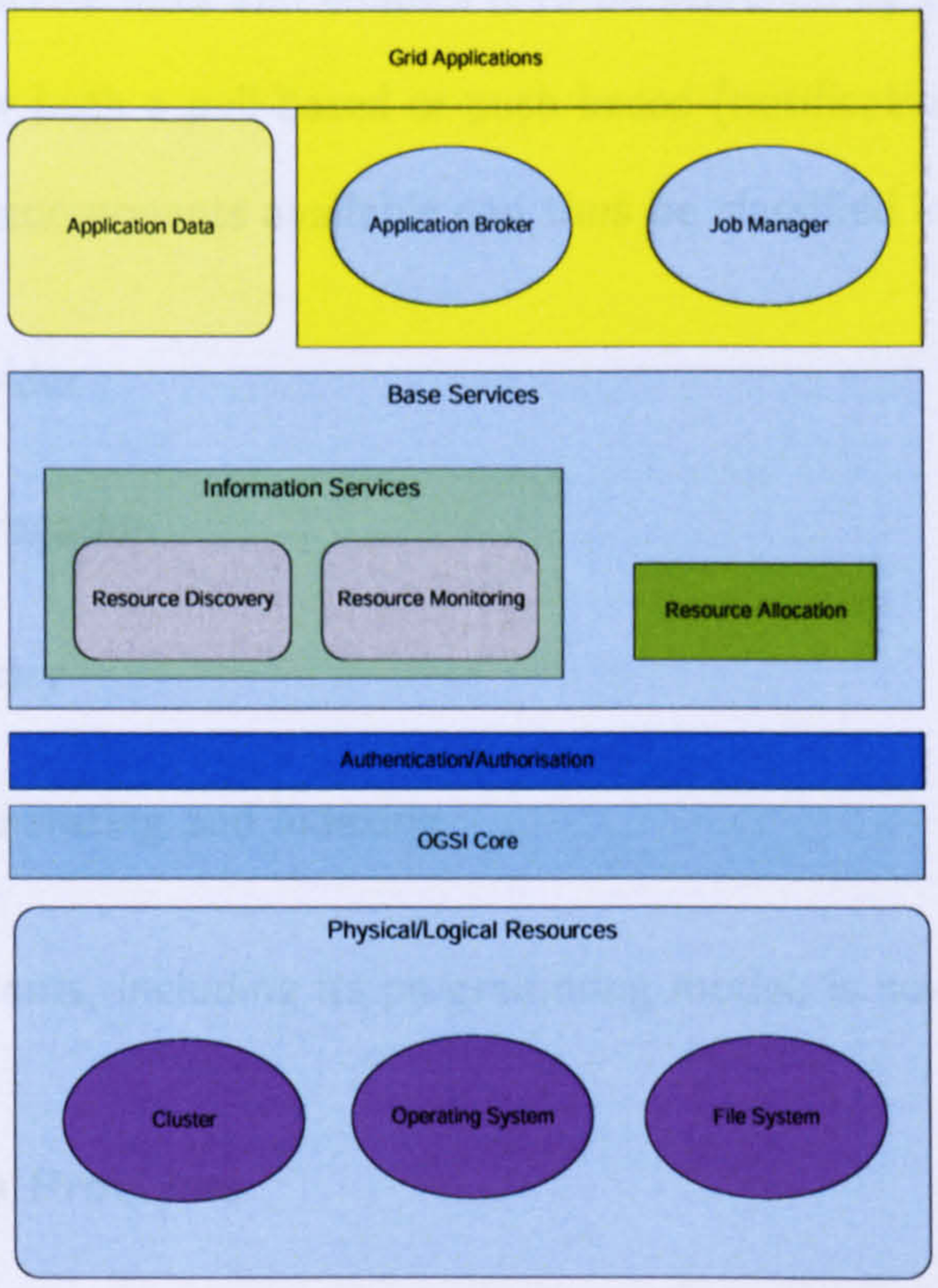

Figure 3.1: The components of Globus Toolkit 3 in relation to Grid applications and their data.

can allow more than one instance. *Service Data Elements* (SDEs) provide a standardised way by which Grid services can convey their state. This is the service data component model on which MDS3 is built. This component model allows the management of complex software using smaller, more manageable modules via the utilisation of the C or Java programming language. The GT3 *Base* package provides the basic GT3 *Core* services as well as high-level services including the Index Service, GRAM (Globus Resource Allocation Manager), Reliable File Transfer and GridFTP (Grid File Transfer Protocol) . GT3 Base offers several categories of information services which use OGSI-based components. Likewise, the component model which the GT3 framework provides, can be used to develop and deliver information services.

The component model for Grid information services offers a standard mechanism for

creating service data exposed from Grid services and other external applications. Moreover, it aggregates all the service data and enables it to be searched by type. The component model also provides for both a pull-based or push-based (notification) query methods on the service data. The components available can thus be classified into the following:

- service data provider
- service data aggregation
- Grid service registry
- dynamic data-generating and indexing

Each of these components, including its programming model, is now described.

3.2.1 Service Data Providers

Service Data Providers create and collect data which can then be used by Grid service instances. The format for the generated data is XML, and the data is represented as a *Service Data Element* (SDE) which can be either a Java output stream or a memory-bound Java DOM (Document Object Model). Moreover, Service Data Provider components are generated by the *ServiceDataProviderManager* Java class and one or more plug-in *ServiceDataProvider* classes. These classes are regularly executed by the Service Data Provider Manager using Java TimerTasks [57]. These provider plug-in programs can take the form of GT3.2 supplied providers or user-created, custom providers. The MDS therefore includes a set of core Service Data Providers providing platform information and it also allows custom-written Service Data Providers to generate any data a user requires. In addition, the service data providers can support both pull- and push-based modes of execution. For a provider to be understood by the MDS, it must be a Java class that implements at least one of three predefined Java interfaces: *SimpleDataProvider*, *DOMDataProvider*, and *AsyncDataProvider*. It must also generate a well-formed, compatible

XML document as the output of its execution. It should thus be possible for the XML document to be parsed in any environment and using any parsing tool, in the form of either a Java output stream or DOM representation.

The service data provider interfaces are:

SimpleDataProvider This is a synchronous provider that outputs XML in the form of a Java OutputStream [56]. An output stream accepts output bytes and sends them to a sink. The *SimpleDataProvider* is the basic interface that all Service Data Providers must implement and is as follows:

```
public interface SimpleDataProvider
  {
    // Returns the display name of the provider.
    String getName();
    /* Returns a description of the provider's
       functionality. */
    String getDescription();
    /* If the provider has a set of default arguments,
       they can be retrieved with this function. */
    String getDefaultArgs();
    /* The provider should return a string representation
       of the current error, if any. */
    String getErrorString();
    /* Triggers the execution of the provider in order
       to update the provider's internal state,
       sending the output to the specified OutputStream. */
    void run(String args, java.io.OutputStream outStream)
         throws Exception;
  }
```

DOMDataProvider This is similar to the SimpleDataProvider in that it is also a synchronous provider, but it generates XML in a w3c.dom.Document format at runtime. Thus, the DOMDataProvider extends the SimpleDataProvider and is as follows:

```
public interface DOMDataProvider extends
```

```
SimpleDataProvider
  {
    public org.w3c.dom.Document run(String args) throws
       Exception;
  }
```

AsyncDataProvider This is an asynchronous version of the SimpleDataProvider that allows for push-based delivery of data. The AsyncDataProvider can also support both the OutputStream object and a DOM document. To use this interface, both the name of a callback function and a valid *ServiceDataProviderDocumentCallback* object must be sent to the *run* method. The *context* parameter is also sent to the *run* method and is used by the caller to pass state information or object references between the calling thread and the callback thread. This provider is as follows:

```
public interface AsyncDataProvider extends
SimpleDataProvider
  {
    /* Triggers the asynchronous execution of the
       provider, which will call the callbackName
       method on the specified
       ServiceDataProviderDocumentCallback object.
       Context is defined by the calling thread. */
    void run(String args,
             String callbackName,
             ServiceDataProviderDocumentCallback
             callback,
             Object context) throws Exception;

/* Signals the provider to shut down, cease
   data callbacks, and free any associated
   resources. */
void terminate() throws Exception;

// Retrieve the current state
```

```
    int getState();

    // provider states
    public static final int PROVIDER_IDLE = 0;
    public static final int PROVIDER_RUNNING = 1;
    public static final int PROVIDER_ERROR = -1;
    public static final int PROVIDER_TERMINATED = -2;
}
```

```
public interface ServiceDataProviderDocumentCallback
  {
    public class[] getCallbackParamSig(String
       methodName);
    public String getDefaultCallbackMethodName();
  }
```

Input to a Service Data Provider

The *run* method which executes a Service Data Provider, can accept a number of string arguments as input. This is done by passing the *serviceDataProviderArgs* member of the *ServiceDataProviderExecutionType* structure to the *executeProvider* port type method for the provider. Furthermore, the *getDefaultArgs* method can be used to access a default list of arguments for the provider. For example, the following serialised XML parameters to *executeProvider* form the input to a Service Data Provider:

```
<provider-exec:ServiceDataProviderExecution>
   <provider-exec:serviceDataProviderName>ForkInformation
   </provider-exec:serviceDataProviderName>
<provider-exec:serviceDataProviderImpl>org.globus.ogsa.
   impl.base.providers.servicedata.impl.
   ScriptExecutionProvider
   </provider-exec:serviceDataProviderImpl>
<provider-exec:serviceDataProviderArgs>
   ./etc/globus-gram-fork-provider
</provider-exec:serviceDataProviderArgs>
```

```
    <provider-exec:serviceDataName>ForkInformation</provider
      -exec:serviceDataName>
    <provider-exec:refreshFrequency>30</provider-exec:
      refreshFrequency>
    <provider-exec:async>false</provider-exec:async>
</provider-exec:ServiceDataProviderExecution>
```

Output from a Service Data Provider

A Service Data Provider outputs XML in the form of either a Java OutputStream or a Java *org.w3c.dom* document. This output becomes the value of a *Service Data Element* for a hosting environment. This SDE can then be used in various ways including for querying and aggregation. For instance, the following is the XML produced when the Simple System Information Provider is executed.

```
<?xml version="1.0" encoding="UTF-8"?>
<mds:Host xmlns:mds="http://glue.base.ogsa.globus.org/ce/1.1"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  ogsi:goodFrom="2003-06-26T02:57:26.296Z"
  ogsi:goodUntil="2003-06-26T03:17:26.296Z"
  ogsi:availableUntil="2003-06-26T03:17:26.296Z"
  mds:Name="localhost" mds:UniqueID="localhost"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="mds:HostType">
<mds:OperatingSystem ogsi:goodFrom="2003-06-26T02:57:26.375Z"
  ogsi:goodUntil="2003-06-26T03:17:26.375Z"
  ogsi:availableUntil="2003-06-26T03:17:26.375Z" mds:Name=
  "Windows XP" mds:Version="5.1" mds:Architecture="x86"/>
<mds:Processor ogsi:goodFrom="2003-06-26T02:57:26.375Z"
  ogsi:goodUntil="2003-06-26T03:17:26.375Z"
  ogsi:availableUntil="2003-06-26T03:17:26.375Z"/>
<mds:MainMemory ogsi:goodFrom="2003-06-26T02:57:26.375Z"
  ogsi:goodUntil="2003-06-26T03:17:26.375Z"
```

```
  ogsi:availableUntil="2003-06-26T03:17:26.375Z"
  mds:RAMSize="2031616" mds:RAMAvailable="574768"
  mds:VirtualSize="2031616" mds:VirtualAvailable="574768"/>
<mds:FileSystem ogsi:goodFrom="2003-06-26T02:57:28.968Z"
  ogsi:goodUntil="2003-06-26T03:17:28.968Z"
  ogsi:availableUntil="2003-06-26T03:17:28.968Z" mds:Name="A:\"/>
<mds:FileSystem ogsi:goodFrom="2003-06-26T02:57:28.968Z"
  ogsi:goodUntil="2003-06-26T03:17:28.968Z"
  ogsi:availableUntil="2003-06-26T03:17:28.968Z" mds:Name="C:\"/>
<mds:FileSystem ogsi:goodFrom="2003-06-26T02:57:28.968Z"
  ogsi:goodUntil="2003-06-26T03:17:28.968Z"
  ogsi:availableUntil="2003-06-26T03:17:28.968Z" mds:Name="D:\"/>
<mds:FileSystem ogsi:goodFrom="2003-06-26T02:57:28.968Z"
  ogsi:goodUntil="2003-06-26T03:17:28.968Z"
  ogsi:availableUntil="2003-06-26T03:17:28.968Z" mds:Name="E:\"/>
<mds:FileSystem ogsi:goodFrom="2003-06-26T02:57:28.968Z"
  ogsi:goodUntil="2003-06-26T03:17:28.968Z"
  ogsi:availableUntil="2003-06-26T03:17:28.968Z" mds:Name="F:\"/>
</mds:Host>
```

3.2.2 Core GT3.2 Service Data Providers

GT 3.2 offers the following core Service Data Providers:

1. **SimpleSystemInformationProvider** This is a Java-based host information data provider that outputs data including the CPU count, memory statistics, operating system type and logical disk volumes.

2. **HostScriptProvider** This is a group of shell scripts for Unix systems that generate different types of detailed host resource information. These scripts output similar information to the LDIF-based (LDAP Data Interchange Format) pre-web services (Pre-WS) ones but in XML format.

3. **AsyncDocumentProvider** This provider is a utility one which periodically reads an XML document from disk through the AsyncDataProvider interface. Moreover,

this provider is useful when the provider developer does not have an interface to the component generating the data, for example during simulations.

4. **ScriptExecutionProvider** This utility provider offers a simple wrapper for the execution of another program, for example a shell script, that generates the XML document data on its standard output stream.

3.2.3 Service Data Aggregation

The MDS3 can collect service data originating from providers, in different ways to provide varying data views. The service data can then be indexed so that queries can be processed efficiently. Furthermore, the aggregator component is comparable to a server-side notification sink. Thus, not only does it listen for notifications on a service data, it also copies the incoming notification data as local service data elements. The service data aggregation component is implemented as an *operation provider* with these exposed operations: addSubscription, removeSubscription and deliverNotification. An *operation provider* allows a Grid service to be composed of different classes. An example of the aggregation implementation with the extended interfaces is:

```
public class ServiceDataAggregatorImpl implements
        ServiceDataAggregatorPortType,
        OperationProvider,
        NotificationSinkCallback
  {
        ...
public String addSubscription(
        AggregatorSubscriptionType type)
  {
    ...
  }
public void removeSubscription(String
        subscription ID)
```

```
    {
      ...
    }
  public void deliverNotification(
        ExtensibilityType message)
    {
      ...
        }
  }
```

3.2.4 Registry Components

The *registry* maintains a set of available Grid Services which can be registered and periodically updated via soft-state registration. Furthermore, the registry is implemented using the OGSI *ServiceGroup* mechanism. It can also provide lifetime, query and service data aggregation on a given member service.

3.2.5 Dynamic Data-Generating and Indexing

The Index Service can create a dynamic data-generating and indexing node which is conceptually similar to the MDS2 hierarchical Grid Index Information Service (GIIS); this will be explained in detail later in the rest of this chapter. The Index Service does so by combining the *ServiceDataProviderExecution* components with the *DataAggregation* and *ServiceGroup* components. Index Services can then be composed into various topologies which are the basis of virtual organisations (VOs).

3.2.6 Information Model for the Index Service

XML is the base information model used by the Index Service. This information model is useful for representing particular sets of service data associated with Grid service instances. These service data XML constructs can therefore be interpreted by both services and

clients. The Index Service information model serves the purpose of providing an interface for the discovery and querying of service data to clients, and the aggregation of service data from heterogeneous Grid services.

In brief, the main function of the Index Service is to specify an interface to generate, access, query and aggregate service data associated with each Grid service. Typically, the information model is composed of several persistent services with more transient services. The OGSI-defined GridService interface exposed by the Grid services allows the two operations: *findServiceData* and *setServiceData* which access and query the service data respectively. It is also used to control the lifetime of the Grid instance. Moreover, the interface corresponding to WSDL *portTypes*, is used to manage the Grid service instances. The interfaces and constructs which are important for the operation of the Index Service, are described below:

- **Factory**, creates a new Grid service instance via its *CreateService* operation. A Grid Service Handle (GSH) (which is described below), is returned as a result of this creation. The *factory* also maintains a searchable set of Service Data Elements.

- **Grid Service Handle (GSH)**, is a unique instance global identifier returned by the Factory `create` operation. A GSH must be converted to a Grid Service Reference (which is described below) before using the service.

- **Grid Service Reference (GSR)**, is the reference to a Grid service, including portTypes exposed by the latter, and it describes how a client communicates with the Grid service instance. Furthermore, the *HandleMap* interface enables a client to map from a GSH to a GSR. The GSH represents name only; on the other hand, the GSR describes a transport protocol and data encoding format through binding information.

- **Query**, allows a service to be queried for service data, via extensible query language support. A Grid service instance maintains a set of Service Data Elements which can

be queried using the *findServiceData* operation from the GridService interface. The queries can be of different types, depending on the Grid service and service container characteristics. In addition, the *findServiceData* operation is a standard, extensible query operation against a service's Service Data Elements, using either the default "by service data name" query or a more complex language like XPath or XQuery. An example client to the query interface is the ogsi-find-service-data command.

- **Registry**, provides a common repository for Grid services via the soft-state registration of those services. A group of Grid services can therefore periodically register their GSHs into a *registry* service to allow for the dynamic discovery of services from that group. This is done by returning the GSHs of the particular set of Grid services.

- **Notification**, provides a dynamic resource state and the delivery of notifications of service data changes for which clients have registered an interest. This registration of interest in a particular service is enabled through the *NotificationSource* interface. This is thus the way in which clients subscribe to service data which can take the form of a service data element, or state values generated by a service at runtime. Upon the availability of updates, the *NotificationSource* interface sends notification messages to registered clients. Additionally, the *NotificationSink* interface is implemented by the client and it allows the asynchronous delivery of notification messages. Examples of the use of notifications include the discovery of services, the delivery of application errors and monitoring.

In short, the *Factory* interface creates a Grid service instance, and returns a GSH which globally identifies the service. In order for the service to be used, the GSH is converted into a GSR to include binding information. A *Registry* allows service instances to be identified, thereby facilitating operations including querying and monitoring. Furthermore, the *Notification* interface allows clients to subscribe to service-related events and to receive notification messages.

3.2.7 The Index Service

One of the major components of MDS3 is the Index Service which provides an extensible framework for accessing, aggregating, generating and querying Grid static and dynamic state data. The Index Service belongs to the collective layer of the Grid middleware infrastructure. Moreover, the Index Service allows external programs to be plugged into the framework by dynamically generating and managing service data. These external provider programs can be the core providers that are part of GT3 (these have been described previously in Section 3.2.2) or user-created, custom providers and they are called *Service Data Providers*. To add a new provider, the latter must be registered in the `index-service-config.xml` configuration file. The Index Service also allows the aggregation of service data from other services. Furthermore, a hierarchy of index services can be created via the composition of registries; a registry maintains a set of available Grid services which are periodically updated. A registry can also be queried for service data. Figure 3.2 shows a high-level overview of the Index Service. It can therefore be observed that the important components of the Index Service are: the service data providers, the data aggregators and the registries, all of which have been described in the sections above.

3.2.8 The Index Service and Asynchronous Queries

The Index Service is a persistent Grid service, which has a set of service data elements associated with it. The service data can be accessed using two methods: query and subscription. The first method is pull-based; this section of the chapter concentrates on the second push-based method.

The Index Service implements the *NotificationSource* interface which allows clients to register their interest in a service by subscription. The source then sends notification messages about the registered data. Moreover, the *NotificationSink* interface enables the Index Service to deliver notification messages asynchronously. Clients register with the

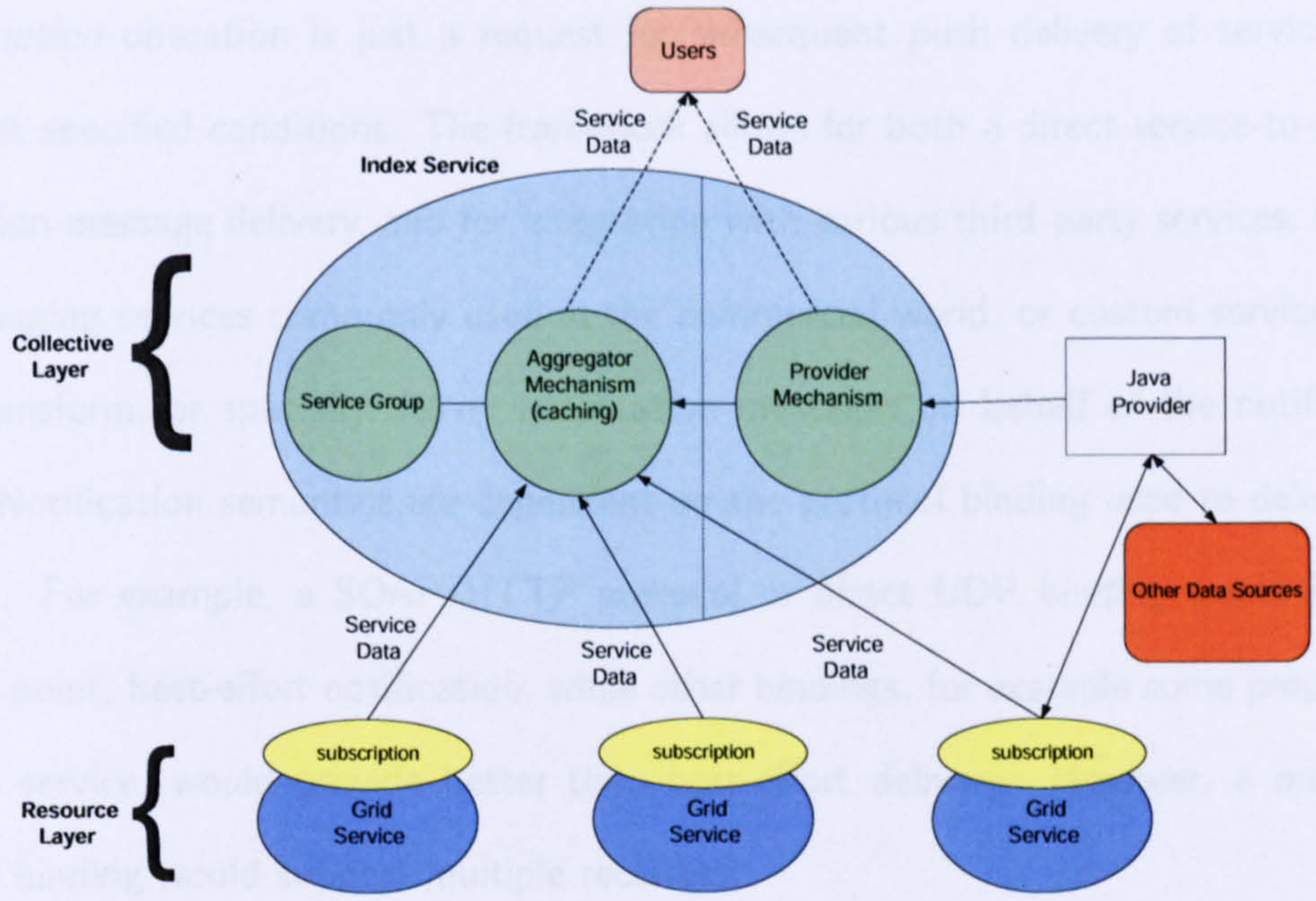

Figure 3.2: Components of the OGSA-based Globus Toolkit MDS3.

service data produced by a service data provider, whose changes are notified by the Index Service.

The OGSA notification framework allows clients to register interest in being notified of particular messages (the *NotificationSource* interface) and supports asynchronous, one-way delivery of such notifications (*NotificationSink*). If a particular service wishes to support subscription of notification messages, it must support the *NotificationSource* interface to manage the subscriptions. A service that wishes to receive notification messages must implement the *NotificationSink* interface, which is used to deliver notification messages. To start notification from a particular service, one invokes the *subscribe* operation on the notification source interface, giving it the service GSH of the notification sink. A stream of notification messages consequently flows from the source to the sink, while the sink sends periodic keepalive messages to notify the source that it is still interested in receiving notifications. If reliable delivery is desired, this behaviour can be implemented by defining an appropriate protocol binding for this service.

An important aspect of this notification model is its close integration with service data: a *subscription* operation is just a request for subsequent push delivery of service data that meet specified conditions. The framework allows for both a direct service-to-service notification message delivery, and for integration with various third-party services, including messaging services commonly used in the commercial world, or custom services that filter, transform, or specially deliver notification messages on behalf of the notification source. Notification semantics are dependent on the protocol binding used to deliver the message. For example, a SOAP/HTTP protocol or direct UDP binding would provide point-to-point, best-effort notification, while other bindings, for example some proprietary message service, would provide better than best-effort delivery. However, a multicast protocol binding would support multiple receivers.

3.2.9 The Index Service and Synchronous Queries

The *findServiceData* operation of the GridService interface is used to perform a query on an aggregated view of service data. This synchronous query method is also known as a *pull* method. Grid services provide support for custom-specified query execution engine. Any number of query expression evaluators can be added by using the query engine functionalities. Examples of some of the expression evaluators supported in GT3 are:

- **ServiceDataNameEvaluator** By using the service data element names, this evaluator allows the client to call the service to obtain service data values. The client subsequently invokes a `findServiceData` operation to initiate the query execution process.

- **ServiceDataNameSetEvaluator** Via this evaluator, the client is able to call the service to set values for service data elements identified by their QName. *QName* [83] represents XML qualified names which consist of a prefix and a local part. Con-

straints including minOccurs, maxOccurs and mutability also influence the addition of elements to a service data value. To start this query execution process, the client invokes a `setServiceData` operation.

- **ServiceDataNameDeleteEvaluator** This evaluator allows the client to call the service to delete service data elements which have been identified by their QName. To start the query evaluation process, the client invokes a `setServiceData` operation.

- **ServiceDataXPathEvaluator** Clients are able to call the service to evaluate XPath expressions on service data elements, through this complex evaluator. The client invokes a `findServiceData` operation to initiate the query evaluation process.

3.2.10 Resource Information Provider Service

The Resource Information Provider Service (RIPS) is a specialised notification service and it forms part of the Globus Resource Allocation Manager (GRAM), as shown in Figures 3.3 and 3.4. This provider service executes system-level scripts and provides tools to monitor forked processes, scheduled queues, file systems and resource statistics. GRAM clients then subscribe to RIPS for notification on job state changes.

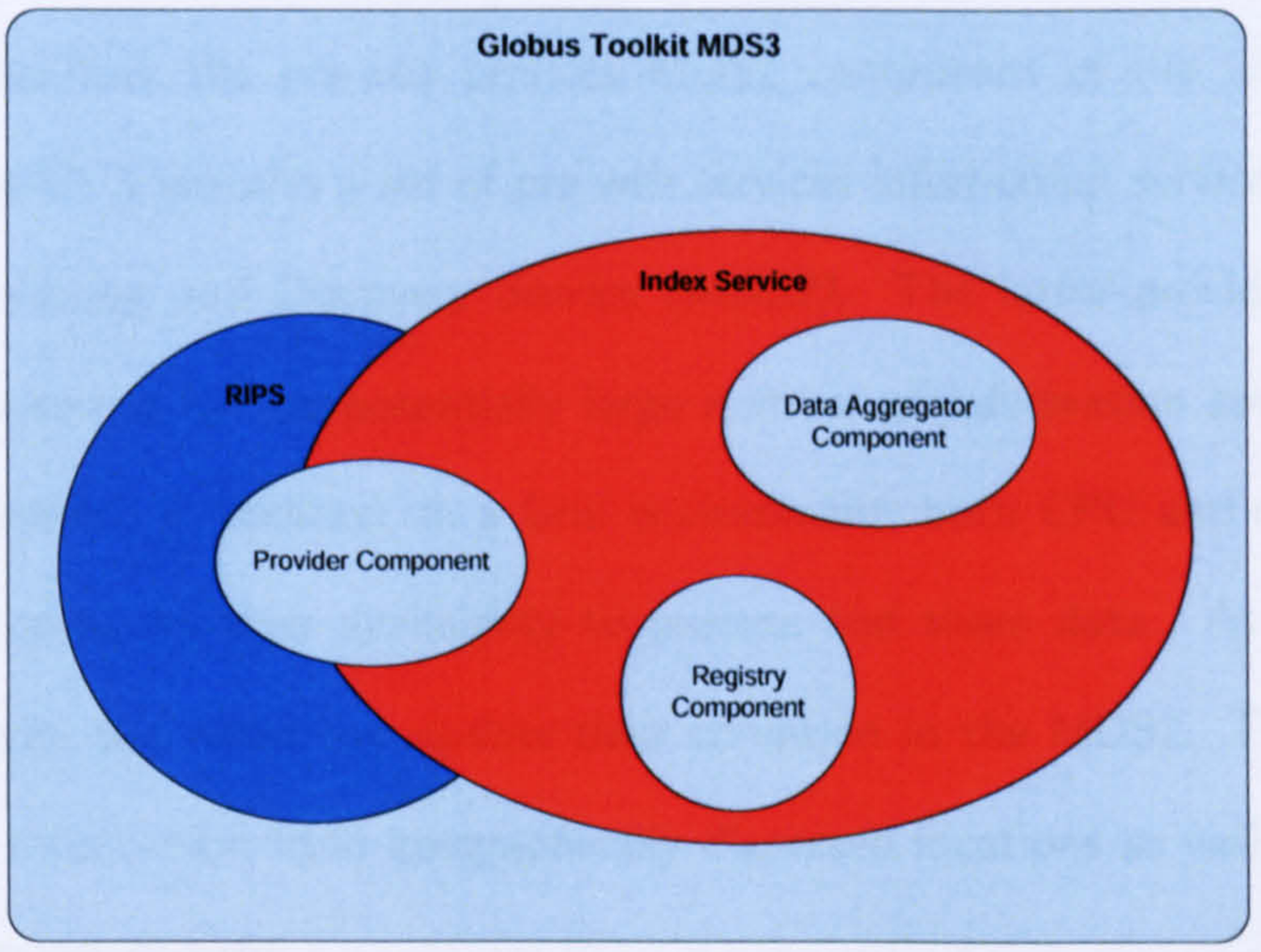

Figure 3.3: Components of the Index Service in relation to RIPS.

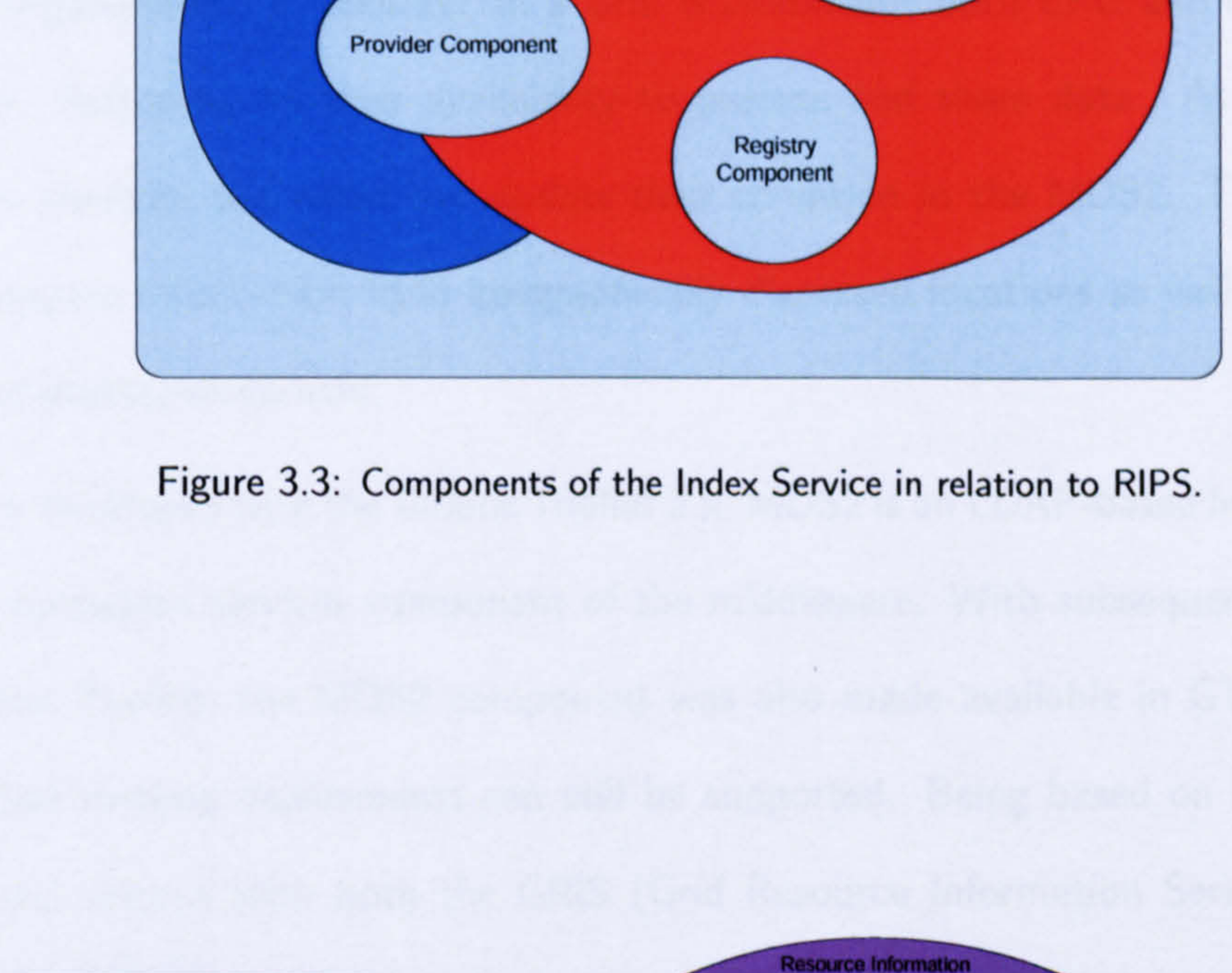

Figure 3.4: Interactions of RIPS and other MDS3 components and schedulers.

3.3 The Monitoring and Discovery Service - MDS2

This section describes the pre-web services MDS2 component of the Globus Toolkit. The Globus Toolkit 3 includes a set of pre-web services information service components, called the *Monitoring and Discovery Service (MDS2)*. The latter provides a standard interface and schemas for the potentially large number of information sources within a Virtual Organisation. In addition, in a Grid environment, both CPU and data resources fluctuate, depending on their availability to process and share data. As the status of resources changes, the latter can update their condition in the MDS2. The MDS2 can also aggregate information from geographically dispersed locations as well as within the same administrative domain.

Originally developed with the Globus Toolkit 2.x, MDS2 is an LDAP-based implementation of the information services component of the middleware. With subsequent upgrades to the Globus Toolkit, the MDS2 component was also made available in GT 3.0, 3.2 and 4.0 so that existing deployments can still be supported. Being based on OpenLDAP, it implements referral with both the GRIS (Grid Resource Information Service) and GIIS (Grid Index Information Service) listening on port 2135. Queries are created of the type `Mds-VO-name` where `Mds-VO-name=local` refers to a GRIS and any other name refers to a GIIS of the same name.

MDS2 [42] provides uniform access to the large number of Grid Information Services found within a virtual organisation, via a single standard interface and schema. It can aggregate information from multiple systems within a single administrative domain, as well as from multiple physical sites. Additionally, MDS2, of which MDS2.4 is the current latest version, offers a standard mechanism for the publication and discovery of resource status and configuration information. It provides a uniform, flexible interface to access data collected by lower-level information providers. Focusing on a decentralised architecture which promotes scalability, MDS2 manages both static and dynamic data. Moreover,

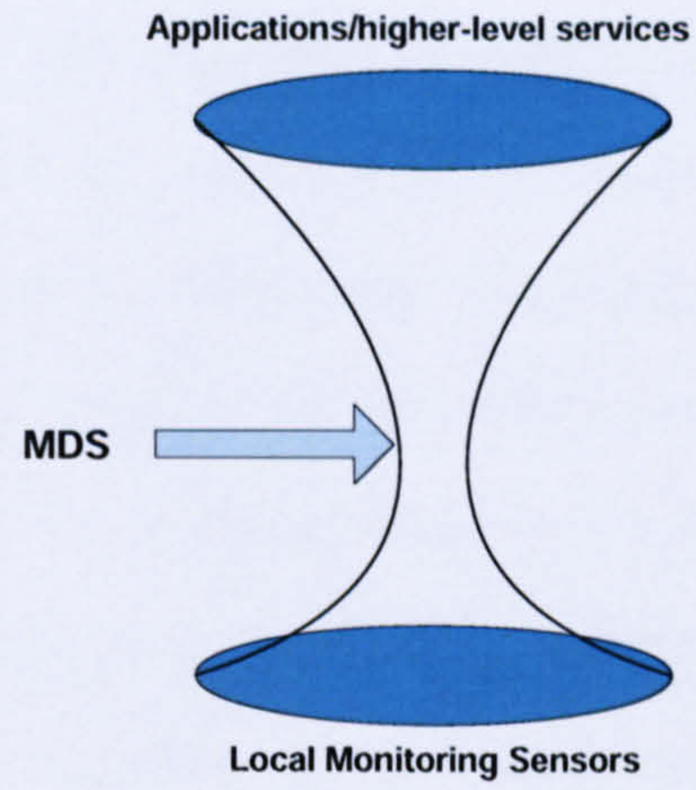

Figure 3.5: The MDS facilitates interactions between local monitoring mechanisms and higher-level services and applications.

access to data can be restricted with the use of both GSI (Grid Security Infrastructure) credentials and the authorisation features provided by the MDS.

The MDS allows uniform access to disparate resource information sources which may use differing information-generating mechanisms. Furthermore, the MDS enables system developers to provide new information services more easily. The MDS2 architecture follows the *hourglass* model that is adopted by most of the components of GT2. As shown in Figure 3.5, the MDS which is the middleware component, is at the neck of the hourglass, with applications and higher-level services at the top level, and localised resource information at the lower level. Therefore, the MDS acts as the intermediary interface between the two levels, whilst decreasing the number of interactions, APIs and protocols required.

MDS2 uses LDAP-based standard mechanisms for publishing and discovering resource status and configuration information. It collects data with lower-level information providers, and through its decentralised structure, it is designed with the intention of being scalable. Moreover, the benefits of the MDS2 is the uniform access to aggregated information, and the ease of adding new information providers.

The MDS has a hierarchical structure which is composed of three main components: the *GRIS*, the *GIIS* and *information providers*. While the *Grid Index Information Service (GIIS)* aggregates lower-level data, the *Grid Resource Information Service (GRIS)* is associated with a resource on which it runs and provides a modular mechanism for the access of information from that resource. Additionally, *information providers (IPs)* represent the source of information itself and interface to a GRIS. In the MDS2 hierarchy, one GRIS could obtain information from one or more information providers. It is usual for a GRIS to register with a GIIS, to allow information from a local site to be accessible at an external site. Moreover, a GIIS can register with another higher-level GIIS using a soft-state protocol which enables resources to join and leave the MDS dynamically. Furthermore, both the GIIS and GRIS allow the caching of information, thereby reducing both the transfer of data and the network traffic.

Resource characteristics can be classified as static and dynamic. Examples of static characteristics include the number of processors, the host model, machine architecture and the operating system version. Dynamic characteristics include CPU availability, CPU load, amount of memory available, usable processors and network load. The Globus Monitoring and Discovery Service uses the LDAP model [50, 118] and represents information in a hierarchical fashion, resulting in a Directory Information Tree (DIT). The MDS architecture consists of these basic entities:

1. Local information from individual configurable information providers is collected in a repository called a **Grid Resource Information Service (GRIS)**. The GRIS also adheres to LDAP and provides information about individual entities. An entity is characterised by a set of *objects* comprised of typed attribute-value pairs. The local information maintained by the GRIS, is updated when requested, and cached for a period of time known as the *time-to-live (TTL)*. The information will time out and will be deleted if no request for it is received by the GRIS. When a subsequent

request is received, the GRIS will launch the appropriate information providers to retrieve the latest information.

2. A higher-level, configurable aggregate directory component called a **Grid Index Information Service (GIIS)**, collects, manages and indexes information registered by one or more GRIS or other GIIS. The aggregate directory services can implement both generic and specialised views and can also provide searching functions. The GIIS can therefore be regarded as a Grid-wide information server which has a hierarchical nature and its own name. Consequently, clients can specify the name of the specific GIIS node they would like to search for information. Additionally, while a GRIS cannot receive registration requests, a GIIS can accept registration of information from a GRIS.

3. **Information providers** are scripts written in a way which adheres to both the input and output interfaces of the GRIS back-end. They also map the status and properties of local resources to the format defined in the LDAP schema and configuration files. Core information providers are included by default in MDS2; however, other resources can be added by creating specific information providers to transfer their status and properties to the GRIS.

4. An **MDS client** queries the MDS for specific information about resources in the Grid environment. The client is based on the LDAP client command ldapsearch or an alternative API.

Figure 3.6 conceptually represents an overview of the MDS components. As illustrated, the resource information is created by the information provider and it is collected by the GRIS. One or more GRIS registers its local information with the GIIS, which can also register with another GIIS, thereby forming a hierarchy spanning across administrative domains. MDS clients can query resource information either directly from a GRIS for local resources, or a GIIS for Grid-wide resources.

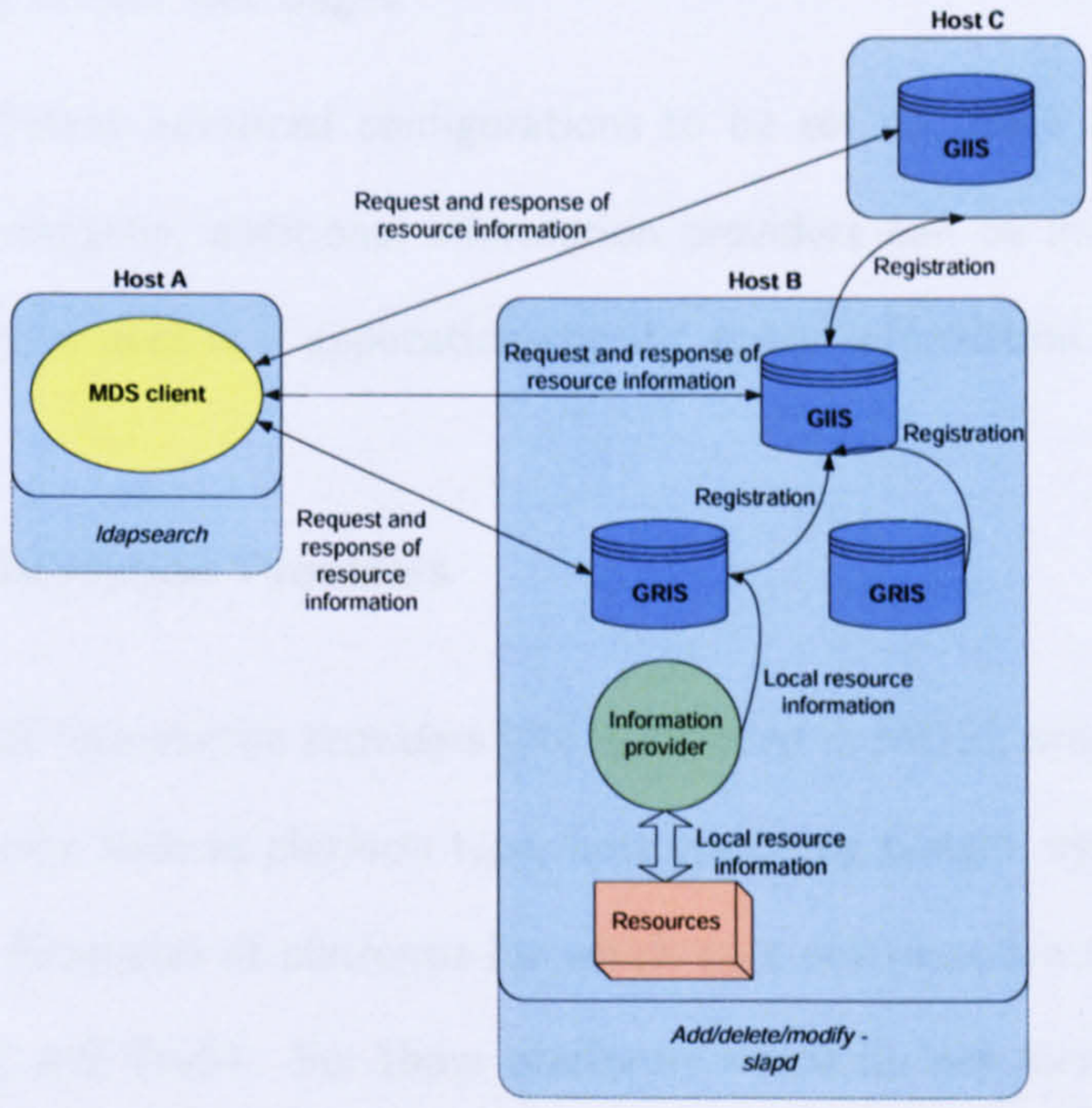

Figure 3.6: An overview of MDS2 components showing the MDS2 architecture as a flexible hierarchy.

The *core* set of MDS2 information providers supply information which includes current load status, CPU configuration, type of operating system and version, file system information, free disk space, the amount of RAM and virtual memory available, and the NIC (Network Interface Card) and type of network connection. Furthermore, information about job status and queue information is provided by the GRAM (Grid Resource Allocation and Management) reporter. The corresponding information providers supply information according to either the MDS core schema or the Grid Laboratory Uniform Environment (GLUE) schema, depending on the configuration setting.

There are several ways in which MDS data can be viewed. For instance, a standard LDAP browser can be utilised to view data from either a GIIS or a GRIS. Moreover, the web-front end can be enhanced using a set of PHP (Hypertext Preprocessor) scripts on a PHP-enabled Web server. The PHP scripts can be added to any web page to perform MDS queries to collect raw information. Additionally, the scripts can be easily adapted to show a project's summary information. Those scripts can also be used to insert MDS

data into existing project web pages.

MDS2 allows different advanced configurations to be set up, once a basic installation is in place. For instance, additional information providers can be included if a cluster-monitoring system is used or if application-specific status information is available.

3.3.1 Core Information Providers

A set of *core GRIS information providers* [80] is included in MDS2 where they are used to generate information such as platform type, host operating system, system load, memory and file system. Examples of platforms for which core providers are available are Linux, Solaris, Irix, AIX and Tru64. For those platforms which do not have any specific core information providers, generic providers are available for the purpose of performing initial MDS operability tests on these platforms.

An information provider can be likened to a *sensor* or a *probe*. Such an information provider is executed to generate the necessary information when a GRIS suffers a cache miss during a resource status query. If this is the case, the information provider is on the critical path of the query latency which a client would observe. To solve this problem, information providers are optimised at install time for the particular platform on which the MDS is installed. Furthermore, each information provider generates LDIF output that represents MDS data objects. These objects are required to match the schema which is present in the GRIS *slapd* server. This is because slapd discards non-matching data.

Each of the core information providers generates a specific part of the whole information about a resource. Therefore, a set of information providers can be created for all supported operating systems, utilising only a small number of variants for each provider category. For example, various Unix-flavoured operating systems can share some of the information providers, whilst differing only in the permutation of the variants defined.

There are a number of parameters which the provider tools can accept to complete the

static data configuration, as well as other options to select subsets of data. Static configuration is a method of optimisation which avoids static values being probed at runtime. This is useful for static information including resource name and MDS Directory Information Tree names. Furthermore, the parameters are usually configured in the GRIS back end provider configuration file, allowing the GRIS to invoke the information providers with the following options:

`-dn`	root of all objects,
`-log`	log file, more specifically, the `grid-info-system.log`,
`-probe-cache`	file for exchanging data between scripts,
`-probe-full`	command for generating full data without any cache.

The following options are supported by MDS 2.4 information providers for the formatting of the output objects:

`-dn`	distinguished name for the object,
`-hostobj`	only the top-level platform object is reported,
`-devclassobj`	intermediate `devclass` objects are reported,
`-devobjs`	leaf objects are reported,
`-noobjs`	no objects are reported (silent probe),
`-classify`	this prefixes output lines to support `sort | uniq` merge of single object,
`-validto-secs`	a control timeout for the length of time data is valid,
`-keepto-secs`	a control timeout for the length of time data should be kept.

Appendix B contains a list of information providers and their different categories and operating system variants. As can be seen from Appendix B, the integrated core GRIS

information providers are composed hierarchically as they share some common shell script functionalities.

3.3.2 Custom Information Providers

It is also possible for a user to create new information providers [78, 128] to publish data into the MDS. These *custom information providers* are able to generate information including host statistics, network status, storage or I/O information, and application-specific information. Indeed, any type of relevant information can be published in the Grid MDS.

The data to be published to the MDS, is converted into LDIF objects using data from both the MDS and that provided by the user. Hence, a custom-created information provider is made available to the MDS. Additionally, the input to the information provider can be specified via a configuration file or at run-time through a query, with several arbitrary command line options.

In MDS2, a user-created information provider is required to generate LDIF objects as the output of its execution. LDIF [110] represents a file format which is used for describing directory information, as well as modifications intended for directory information. Moreover, LDIF is utilised for the exchange of directory information amongst LDAP-based directory servers.

Writing a new information provider involves three main steps as shown below:

1. The first step is to identify the type of information to be published to the MDS. Then, the placement of that information within the Directory Information Tree needs to be decided upon. This process involves the definition of a schema, an Object Identifier (OID) assignment and naming conventions for the items of information.

2. The next step is to develop a program that conforms to the input and output re-

quirements for information providers. It must thus be possible to invoke the program using the `fork()` and `exec()` methods of the GRIS back end, and the program should return LDAP Data Interchange Format (LDIF) data objects as defined by the corresponding schema. The information provider program can be written in any programming language.

3. Since the `grid-info-resource-ldif.conf` file contains a list of all active GRIS providers, an additional entry for the newly created program needs to be added to it. These active GRIS objects specify how the respective information providers should be invoked. Subsequently, the GRIS back end reads the `grid-info-resource-ldif.conf` file to obtain the path name (*path:* and *base:* parameters) and the arguments (*args:* parameter) for the information provider. The GRIS back end consequently *forks* and *execs* the information provider.

3.3.3 OpenLDAP

MDS2 is implemented using OpenLDAP which is an open source package employing the *Lightweight Directory Access Protocol (LDAP)*. It consists of an LDAP server, LDAP replication server, libraries for the LDAP protocol and other utilities, including some sample clients. Working over TCP/IP, LDAP stores information with a unique *Distinguished Name* and associated attributes. Each of these attributes is made up of a value-type pair. For instance, an attribute for a resource may have the type *osname* and value *Unix*. Furthermore, the LDAP entries are stored in a hierarchical structure where logical boundaries apply. For example, the top of the tree structure may represent different virtual organisations, and the branches denote the attributes corresponding to each VO.

Filters can also be used with LDAP queries. This allows only certain sections of the tree structure to be searched, while irrelevant parts are disregarded. Moreover, access to LDAP directory servers can be restricted, resulting in clients having to authenticate their identity

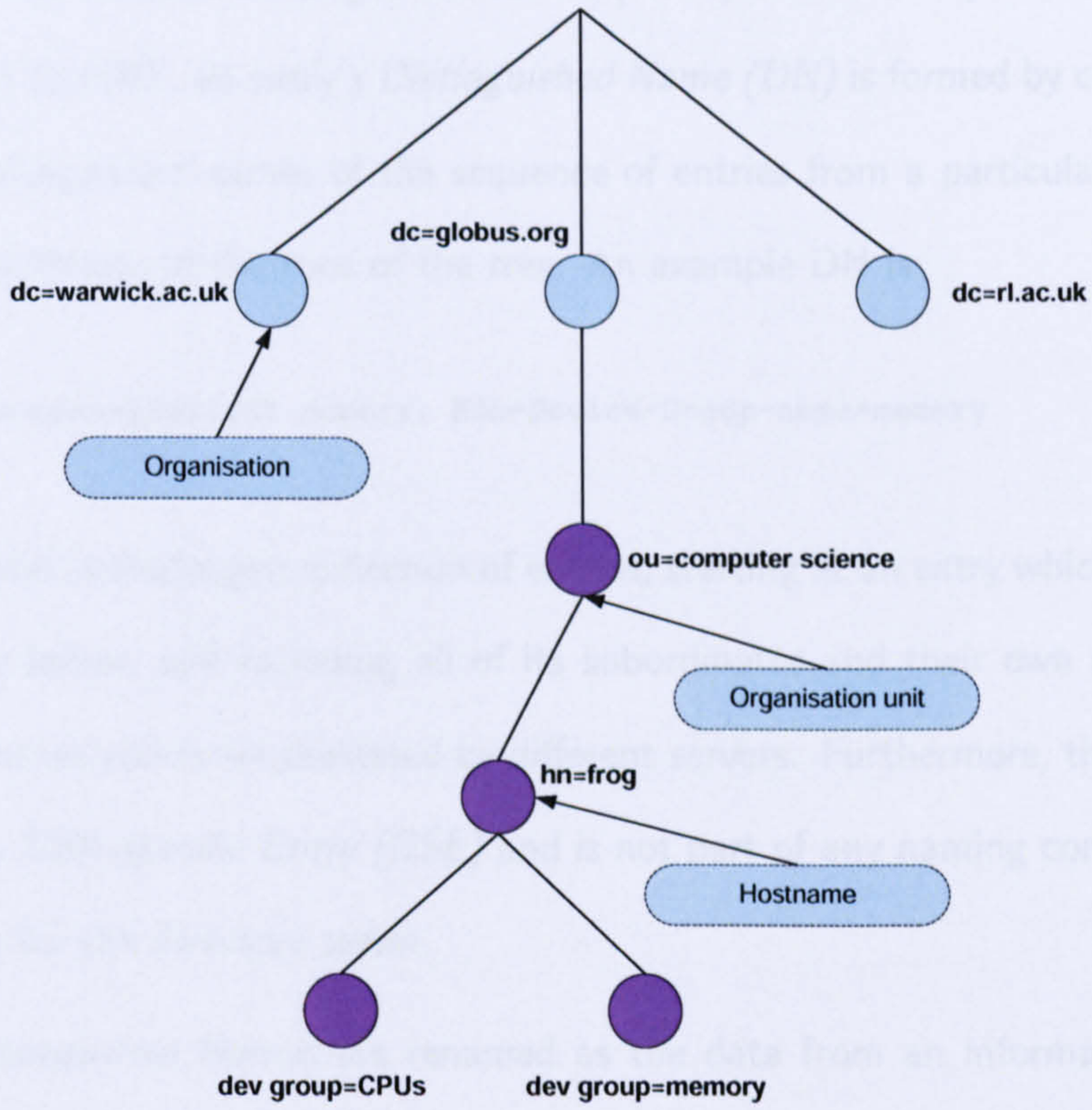

Figure 3.7: LDAP directory tree structure.

first. The LDAP structure is shown in Figure 3.7.

3.3.4 Distinguished Names (DN)

A *directory* consists of entries containing descriptive information which is stored in the form of attributes for the entries. Each attribute describes a specific type of information. Based on a client-server model, LDAP defines a directory service and access to that service. LDAP servers provide the directory service, whilst LDAP clients use the directory service to access entries and attributes. Moreover, data is organised hierarchically, starting at the root and branching downwards into individual entries.

LDAP uses the X.500 data model [64] which assumes that there is one or more servers jointly providing access to a *Directory Information Tree (DIT)*. The DIT is made up of a number of entries that possess names. Moreover, one or more attribute values from the

entry make up its *Relative Distinguished Name (RDN)* which is required to be unique. Being unique in the DIT, an entry's *Distinguished Name (DN)* is formed by concatenating the relative distinguished names of the sequence of entries from a particular entry to an immediate subordinate of the root of the tree. An example DN is:

```
dn: Mds-Device-name=physical memory, Mds-Device-Group-name=memory
```

A *naming context* is the largest collection of entries, starting at an entry which is mastered by a particular server, and including all of its subordinates and their own subordinates, down to the entries which are mastered by different servers. Furthermore, the root of the DIT is called a *DSA-specific Entry (DSE)* and is not part of any naming context. DSA is an X.500 term for the directory server.

In MDS2, Distinguished Names are renamed as the data from an information provider propagates up through a hierarchy of GIIS servers. This allows query functionality to be handled properly [77].

3.3.5 MDS Protocols

Each information provider implements two basic protocols: the GRid Information Protocol (GRIP) which enquires about the structure and state of a resource or service, and the GRid Resource Registration Protocol (GRRP) which allows a resource to both register with another entity and to notify the latter of its availability. The protocol also specifies how to contact the entity for the purposes of enquiry or control.

GRIS adheres to the GRIP protocol, and GIIS to the GRRP [21].

3.3.6 GRIS

This information provider framework is implemented as an OpenLDAP server back-end which is customisable using plug-ins from specific information sources; this is shown

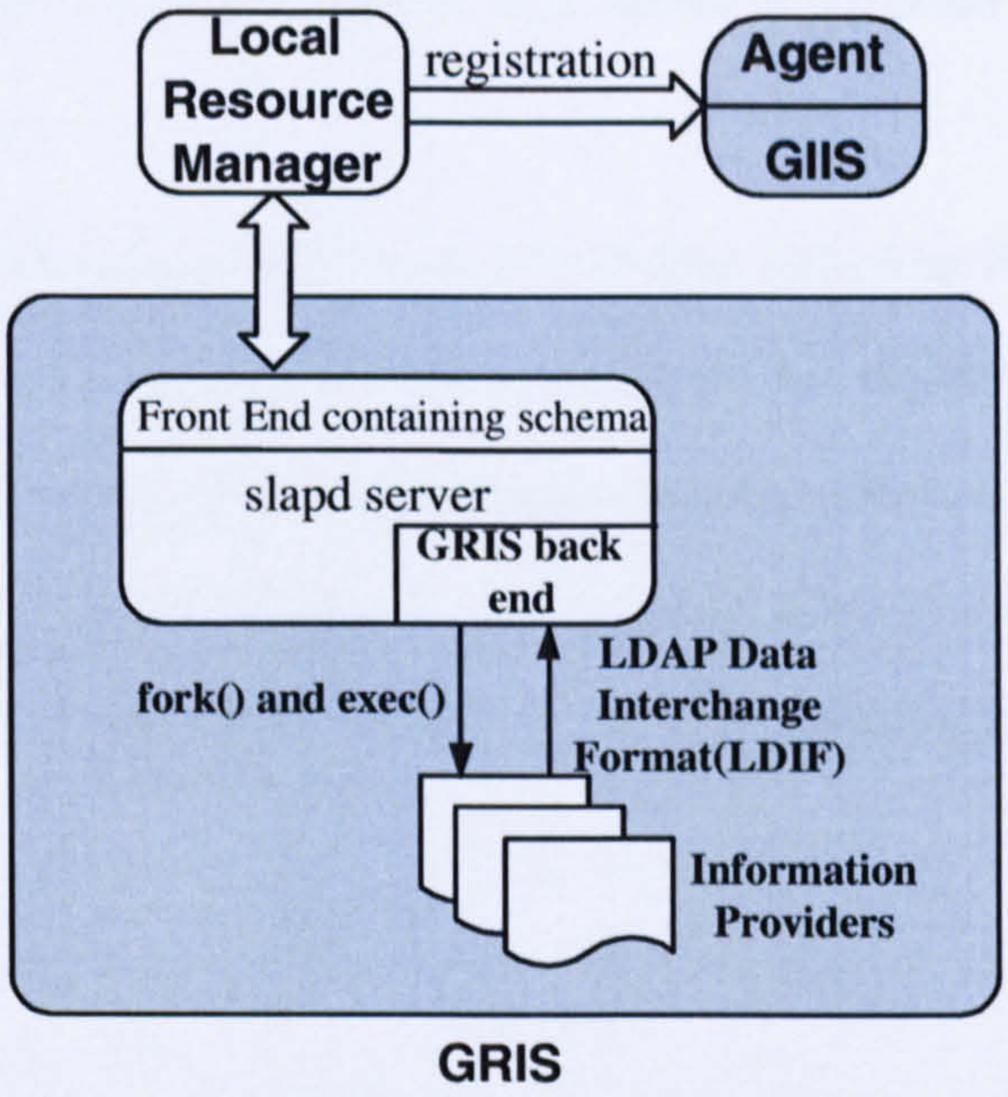

Figure 3.8: Within the GRIS, the server front end contains a schema for the providers. The GRIS back-end is within the LDAP server and it *forks* processes and *execs* provider programs which then return LDIF data to the back-end.

in Figure 3.8. Considering the Grid environment, each resource under local scheduler management can run a local GRIS. A GRIS can service requests for specific resources, but in the figure, a GRIS is configured to register itself with an aggregate directory service (GIIS) via the local resource which can be a scheduler, so that information can be passed onto other end users, represented here as agents.

Typically, a local resource manager would send out an information request to each GRIS under its logical management on a periodic basis (for example, every minute). The GRIS on each resource would then authenticate and parse the incoming information request, and then dispatch the request to be handled by a local information provider. The local resource manager then merges the results from each of its GRIS resources and pushes them to the aggregate directory service.

The communication between a GRIS and an information provider takes place over a well-defined API. The local resource manager also caches each information provider's results for a specified period of time. Thus, the number of provider invocations is greatly reduced, the response time is improved and deployment capability is maximised. This configurable

length of time is the cache's time-to-live (TTL), which is specified for each information provider at configuration.

The information providers return static host information including the number of processors, the CPU model, the operating system version and the architecture type. Dynamic host information is also pulled, including the load average, CPU availability, the number of available processors, storage information and network information. The local resource manager then pushes the information to the agent's GIIS, when requested.

3.3.7 GIIS

The MDS also provides a framework for constructing aggregate directories called Grid Index Information Services (GIIS). GRRP messages are passed from 'child' GRIS to the directory to form a unified information repository. The three major components making up the GIIS framework are:

1. Generic GRRP handling,

2. Pluggable index construction,

3. Pluggable search handling.

The local resource manager pulls information from its information providers via its *Resource Monitoring* component and pushes that information to the agent's GIIS via its *Resource Information Reporting* module. The GRIS component is made up of the information providers and the *Resource Monitoring* module.

3.3.8 MDS Configuration Files

To create a hierarchical GIIS architecture, it is important to first decide on the hierarchical structure of GIIS servers. Subsequently, several configuration files are created and

modified. These files [79] are located in `$GLOBUS_LOCATION/etc` and their purposes are detailed below.

- **grid-info.conf** This configuration file sets the default values for the arguments to the `grid-info-search` command which allows clients to query a GIIS or a GRIS by specifying a number of options. The `grid-info.conf` file also specifies the MDS administrator's email address.

- **grid-info-resource-ldif.conf** This file allows the specification of active and available GRIS information providers which are able to send data to the GIIS they are registered with. This file describes the core MDS information providers as well as custom-specified ones for the GRIS. It also specifies the set of available providers and the way in which they fit into the hierarchy of *Distinguished Names* (DNs) in the *Directory Information Tree* (DIT).

- **grid-info-resource-register.conf** The GIIS servers to which a GRIS registers are listed in this file. The default situation is for the GRIS to register with the local GIIS on the same host. Moreover, details included in this file are host names, ports, as well as time values for the control of registration messages from a GRIS to a GIIS server. Another item of information specified is the binding method for mutual authentication between both GIIS and GRIS machines, and between GIIS machines in a hierarchy.

- **grid-info-site-giis.conf** This file initialises the data structure used by a GRIS registering to a GIIS. After the GIIS server reads this file, it initialises registration entries in its own data structure. This is carried out independently of registration messages from other GIIS or GRIS machines. Additionally, via this file, the GIIS is able to set timing, registration control and binding method parameters. In the absence of such a file, those parameters can only be set by registrants sending registration messages to the GIIS.

- **grid-info-site-policy.conf** The GIIS manages the acceptance of registration messages through this configuration file. This is achieved by creating different policies, for instance, an open policy where all registrants are allowed to register, or a closed one where only certain resources are allowed to register with a GIIS. Moreover, the binding method for mutual authentication between a particular GRIS or GIIS resource registering with a GIIS, can also be specified in this file. The default configuration is for the GIIS to accept registrations both from its own server and from port 2135.

- **grid-info-slapd.conf** This file allows the GIIS and GRIS provider components to be specified in OpenLDAP, by setting information access control rules, determining LDAP and MDS information schema, defining the back ends supported by the *slapd* server, and by setting anonymous binding. Other control parameters which the `grid-info-slapd.conf` supports are the number of objects returned by the *slapd* server [90] to the client, the maximum length of time the slapd server should spend in answering each search request, as well as the maximum number of worker threads in a slapd process. However, the drawback of increasing the maximum number of threads is that more resources including memory, is being used up, even though a larger number of concurrent queries can be handled.

- **grid-info-deployment-comments.conf** This file includes an administrator-related comment concerning the MDS system deployment. The comment is editable and is used in the output of the `grid-info-search` command when all the objects on a host are being queried.

- **grid-info-server-env.conf** The values of environment variables are specified in this file when the MDS is started; examples are the certificate and key.

- **gridftp-resource.conf** This file is used in conjunction with the `gridftp-perf-info` information provider which publishes GridFTP performance information to the MDS. Moreover, this file contains information concerning the GridFTP environment

as well as information-reporting requirements in terms of hostname, URL and the location of the logs.

Each of the above configuration files is provided in Appendix C. These files come from a typical MDS2.4 installation; more details are given in the appendix.

3.3.9 MDS Information Provider Schemas

A *schema* [65] represents the information, including attribute type definitions and object class definitions, which an LDAP server uses to determine the way in which to match a filter or attribute assertion (in a *compare* operation) against the attributes of an entry, and whether to allow *add* and *modify* operations. A directory schema is thus a set of rules which defines how data can be stored in the directory, in the form of entries. Each *entry* is a set of attributes and their values, and it should have an *object class*. The object class specifies the type of object which the entry describes and defines the set of attributes it contains. The schema defines the type of entries allowed, their attribute structure and the syntax of the attributes.

The attribute types are described by sample values for the subschema *attributeTypes* attribute, which is written in the *AttributeTypeDescription* syntax. Furthermore, an entry for each object class contains an abstract class (`top` or `alias`), at least one structural object class, and zero or more auxiliary object classes. It is at the time of assignment of the object class identifier that the object class is defined as *abstract*, *structural* or *auxiliary*.

- **Abstract** Used to derive other object classes, this object class is a superclass or template which collects a set of attributes common to a set of structural object classes. However, LDAP entries cannot belong to an abstract object class, and must instead belong to a *structural* object class.

- **Structural** A structural object class indicates the attributes which an entry may have and where each entry may occur in the Directory Information Tree. It is a requirement that entries should belong to a structural object class; therefore, most object classes are structural object classes.

- **Auxiliary** This type of object class defines the attributes which an entry might have. An auxiliary object class represents additional attributes that can be associated with a structural object class as an addition to its specification. Each entry may belong to only a single structural object class, or to zero or more auxiliary object classes.

The LDAP server uses *matching rules* to compare attribute values with assertion values during *search* and *compare* operations. Furthermore, the default directory schema can be extended with new object classes and attributes. A new object class is created to contain new attributes which are added to the schema.

In MDS 2.4, the physical and logical components of a compute resource are modelled as a hierarchy of elements. A number of MDS element types exist, which correspond to LDAP structural `objectclasses`. Some examples include:

```
class MdsVo
    contains attr Mds-Vo-name
  class MdsHost
    contains attr Mds-Host-hn
  class MdsDevice
    contains attr Mds-Device-name
class MdsDeviceGroup
  contains attr Mds-Device-Group-name
```

Additionally, a set of auxiliary object classes exist, providing complementary information about the particular elemental instances. The MDS 2.4 information model [81] uses this LDAP characteristic to merge objects with information which is higher in the object tree.

Therefore, while a *leaf* node contains information about a single resource instance, a *parent* node contains the merged information about several instances. Examples are:

```
dn: Mds-Device-Group-name=memory, ...
    objectclass: MdsMemoryRamTotal
    objectclass: MdsMemoryVmTotal
    objectclass: MdsDeviceGroup
    Mds-Device-Group-name: memory
    Mds-validfrom: 200110030128.12Z
    Mds-validto: 200110030128.12Z
    Mds-keepto: 200110030128.12Z
    Mds-Memory-Ram-Total-sizeMB: 751
    Mds-Memory-Ram-Total-freeMB: 642
    Mds-Memory-Vm-Total-sizeMB: 1600
    Mds-Memory-Vm-Total-freeMB: 1592
    Mds-Memory-Ram-sizeMB: 751
    Mds-Memory-Ram-freeMB: 642
    Mds-Memory-Vm-sizeMB: 1600
    Mds-Memory-Vm-freeMB: 1592

dn: Mds-Device-name=physical memory, Mds-Device-Group-name=memory, ...
    objectclass: Mds
    objectclass: MdsDevice
    objectclass: MdsMemoryRam
    Mds-Device-name: physical memory
    Mds-Memory-Ram-sizeMB: 751
    Mds-Memory-Ram-freeMB: 642
    Mds-validfrom: 200110030128.12Z
    Mds-validto: 200110030128.12Z
    Mds-keepto: 200110030128.12Z
```

This feature of merging multiple types enables the parent object to classify the children objects, as well as reflect the different types of each child node. The advantage of the merged object is the expression of constraints on multiple data, in search filters. However, certain information is lost through the inability of the LDAP data model to

distinguish particular instances of an attribute value. The `MdsDeviceGroup` object names `processors`, `memory`, `filesystems` and `networks` are groupings for the instances of the corresponding devices in those categories. Moreover, the `MdsSoftwareDeployment` object name `operating system` references information about the bootable operating system software available on the resource.

3.3.10 GLUE Schema

The objective of the *GLUE* schema [44, 46] is to define an abstract information model and for the mapping to concrete schemas for the representation of Grid resources. This is done to describe Grid resources precisely and systematically for subsequent discovery and management. The GLUE schema, also called the *GLUE Information Model*, was first developed as a collaboration effort by the EU-DataTAG [108] and US-iVDGL [52] projects. Further projects which are participating in this effort are EGEE, LCG, Grid3/OSG, Globus and NorduGrid.

The GLUE schema describes core Grid resources at the conceptual level by defining an information model which abstracts real world objects into constructs. Examples of these constructs are objects, properties, behaviour and relationships. The main advantage of the GLUE schema is that it does not depend on any particular implementation and can thus be used to exchange information amongst different knowledge domains. Moreover, this information model can also be mapped onto data models which are specific to particular Grid information services.

A core concept of the information model used for abstracting computing resources managed by different local resource managers, is the *Computing Element (CE)*. It describes the computing service which is offered at the virtual level to group of users or Virtual Organisations. Locally managed computing resources including the Portable Batch System (PBS), Load Sharing Facility (LSF) or Condor, all have differing capabilities but yet have

similar characteristics, which include scheduling functionalities using queues, and sets of policies. A common method of abstracting these different systems is to have a Computing Element refer to the characteristics, resource set and policies of a single queue of the underlying management system. Subsequently, computing capabilities uniformly appear as Computing Elements which are reachable from a specific network endpoint. Since local resource managers can be configured to assign group-specific elements to queues, different groups of users have different views for a CE. Additionally, the *VOView* entity allows different states to be modelled for various groups of users.

Furthermore, the GLUE schema enables the abstraction of storage resources. A range of storage resources contribute to the Grid, from basic disk servers to complex, massive storage systems. Different services manage these resources, handling functionalities including data access, quota management and space management. The GLUE schema provides the *Storage Element SE* as a concept for identifying the various services which are responsible for managing the storage resources. The concept `Storage Area` which is assigned to a group of users or VO, allows the abstraction of the storage resource.

The GLUE schema also defines the relationships that exist between Computing and Storage Elements. Different types of such relationships are useful to be discovered from Grid Information Services as they contribute to Grid-level scheduling. Additionally, the modelling of a generic host entity is permitted in the GLUE schema and is used mainly for functional monitoring. Details included in the *Host* entity are architecture, memory, network, load, processor, operating system and file system.

The GLUE schema is interoperable with MDS2 information providers, and in GT3, the GLUE schema is used natively, through its XML mapping [45]. The installation of the GLUE schema is not required in GT3.

3.3.11 Provider Schemas, OIDs (Object Identifiers), and Namespaces

Global services are usually distributed, meaning that the data they contain is spread across many machines, all of which cooperate to provide the directory service. Typically a global service defines a uniform namespace which gives the same view of the data no matter where one is in relation to the data itself. The Internet Domain Name System (DNS) is an example of a globally distributed directory service.

In MDS2, the LDIF objects which an information provider outputs, must correspond to the schema found in the *slapd* server front end. It is the responsibility of the GRIS back end of the *slapd* server to ascertain that any LDIF objects returned, conform to the corresponding search request. These data objects should also conform to LDIF syntax rules. Otherwise, if there is any mismatch, the *slapd* server will suppress the non-matching data and the GRIS will not operate properly.

Moreover, an OID for each information provider must exist in the schema for every class and attribute type. The attribute name and class name have aliases and their own prefixes can be used in a string.

With the increasing number of user-created information providers, it is crucial that these providers have unique, non-conflicting OIDs and names, to avoid overlapping and confusion. Consequently, every provider should be associated with a unique prefix that identifies the organisation from where the provider originates. The latter part of the name can be anything the developer chooses. Furthermore, OID and naming assignment should be coordinated and controlled in order to avoid name collisions completely.

A method for managing those assignments is by using a Private Enterprise Number (PEN) which is issued by the Internet Assigned Numbers Authority (IANA). For example, an OID subspace is obtained from IANA for an organisation. This organisation then utilises a segment of that subspace for the purposes of tracking and controlling OIDs assigned to custom-created information providers. Therefore, by using OIDs, the IANA can ensure

that a single prefix is not used by more than one organisation.

The insertion of the new information provider into the GRIS namespace relies on the schema for the provider. The namespace which is used by the MDS is an OID subspace registered with the Internet Assigned Numbers Authority (IANA). This OID subspace allows OIDs for new custom-created information providers to be monitored and controlled. Additionally, a single prefix should be used for all the various names created within an organisation in order to avoid OID and name collisions with those at other organisations.

The relevant OIDs are:

- 1.3.6.1.4.1 **IANA PEN space**
- 1.3.6.1.4.1.3536.* **Globus OID subspace**
- 1.3.6.1.4.1.3536.2.* **Globus Information Services OID subspace**
- 1.3.6.1.4.1.3536.2.6.* **MDS OID subspace**

The MDS schema is based on LDAP schemas [89] which are used for matching syntaxes, rules, attribute types and object classes [65] on the *slapd* server [90]. The MDS core schema uses the above reserved MDS OID subspace and the name prefix `MDS`; for instance, `MDS-VO-name`.

3.3.12 Registration Control and Handling of Time

Both the GRIS and GIIS are queried on port 2135 and the way of differentiating between them, especially if they are both on the same machine, is by the DN used to set queries. The GRIS has `mds-vo-name=local` while the GIIS has `mds-vo-name=site`. The parameters in the `grid-info-resource-register.conf` file identifies host names, ports and time values which control the registration messages from a GRIS to a GIIS server [77].

One of the control parameters is the *LDAP sizelimit* which resides in both the `grid-info-`

`resource-ldif.conf` and the `grid-info-slapd.conf` files. This parameter defines the maximum result set size of the number of objects which the server can return for any given client request. This parameter should be set appropriately, depending on the type of server and the amount of data expected to be returned from the queries.

Two other parameters residing in the `grid-info-resource-register.conf` file are *regperiod* and *ttl*. The registration period is the notification time for service availability, which a *registrant* sends out to a *registrar* for notification of its existence. The *ttl* parameter specifies the length of time for which the registrar should keep the registration information. A general guideline is for the *ttl* value to be twice the registration period.

Furthermore, the *cachettl* parameter which exists in the `grid-info-resource-register.conf` file, specifies the length of time for which the registrar should keep the registrant's data in its cache. The default value is 30 seconds. The data in the registrar's cache is returned when queries are received by the registrant within the time specified by *cachettl*. Otherwise, if the *cachettl* value has expired, the registrar requests the data from the next lower level in the MDS hierarchy. This procedure is illustrated in Figure 3.9. The diagram only shows a single registrar and a single registrant, as well as one information provider. However, in a GIIS hierarchy with multiple registrars and registrants, a registrant GRIS verifies the cache time of each information provider.

MDS utilises its own caching mechanism and therefore does not rely on that provided by standard OpenLDAP. Additionally, it is important that the system clocks on the machines in the MDS hierarchy are synchronised. Otherwise, registration messages may be lost, partial data may be returned or no data at all.

An option in the `grid-info-search` command, `giisregistrationstatus`, allows the status of servers which are registered to a GIIS or from which a GIIS is receiving data, to be checked. The output from the command show registration objects which include the status type: valid, invalid, or purged. These status types are derived from the `validfrom`,

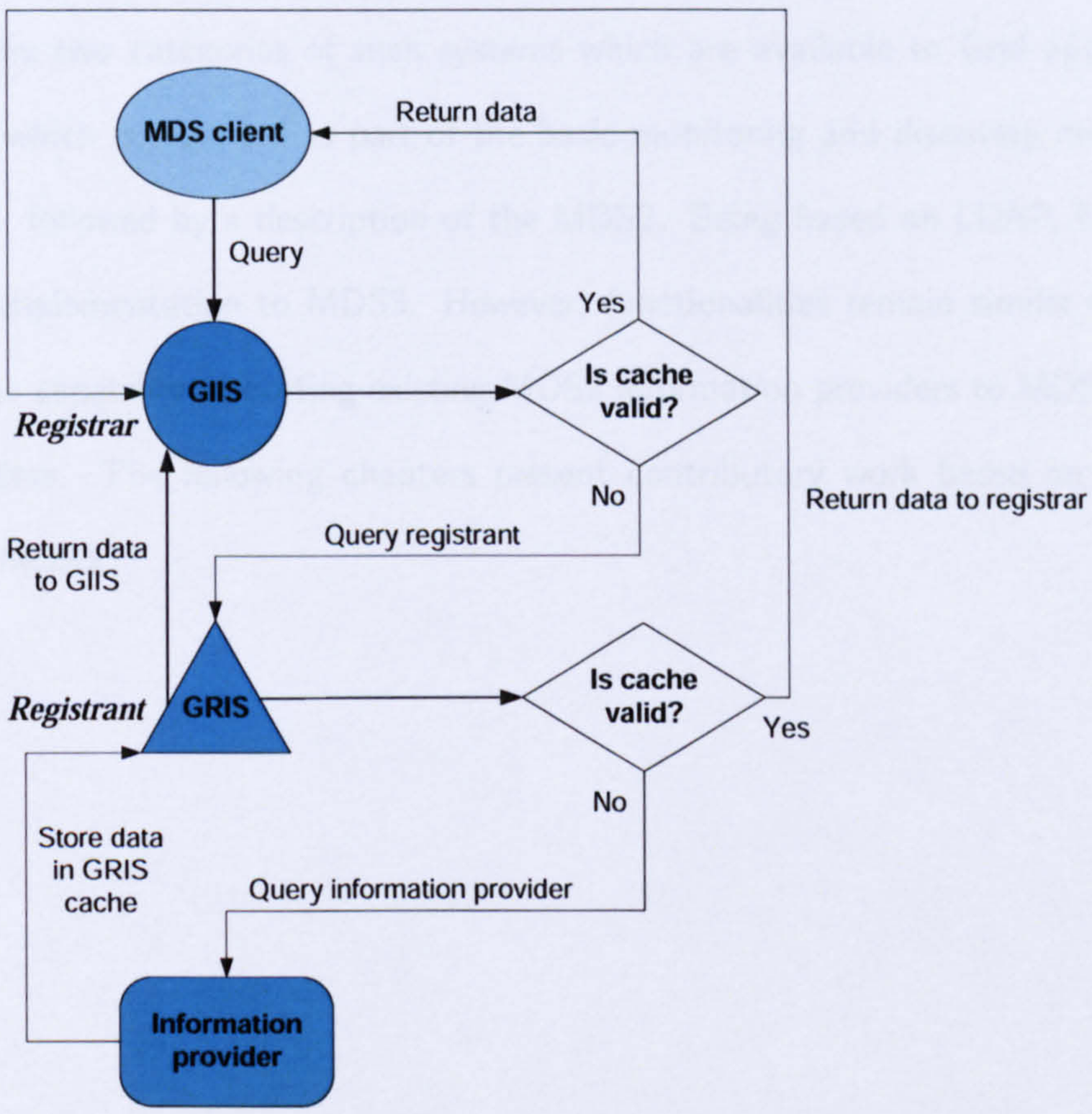

Figure 3.9: The application of the cache time-to-live parameter when returning data to the client from the GIIS and GRIS.

validto and keepto parameters which are generated from the *ttl* parameter in the grid-info-resource-register.conf file. The former parameters represent the time-frame during which a server maintains the registration messages sent from another server.

3.4 Summary

This chapter highlighted the required features for Grid monitoring and discovery systems, as well as the two categories of such systems which are available to Grid applications. The MDS3 which is regarded as part of the basic monitoring and discovery middleware, is presented, followed by a description of the MDS2. Being based on LDAP, MDS2 has a different implementation to MDS3. However, functionalities remain similar with GT3 providing the capability of porting existing MDS2 information providers to MDS3 Service Data Providers. The following chapters present contributory work based on both the MDS2 and MDS3.

Chapter 4

MDS2 Experimental Evaluation

4.1 Introduction to Grid Information Services

One of the characteristics of Grid information services is to gather and manage *dynamic* data, that is, data which has an often short lifetime of utility, or which is updated frequently. However, there are conflicting requirements for Grid information services and no single system completely covers all aspects and conditions. For example, the Grid information service must understand the semantics attached to the data it represents so that adequate expression and flexibility are provided. However, this requirement should not impede on the efficiency of data collection and delivery. It is crucial for Grid information systems to provide high performance because the performance of application resource selectors and schedulers at runtime, directly influence application execution, in the form of overhead. One way for a Grid information system to ensure that the necessary performance goals are met, is for it to support caching. Furthermore, while the Grid information system should provide a unified service and use a universal data representation, it is also required that data can be translated to other formats and encodings with minimal overhead. Subsequently, the Grid information system meets the requirement that the *presentation* of the data should be separated from the *storage* and *retrieval* of data.

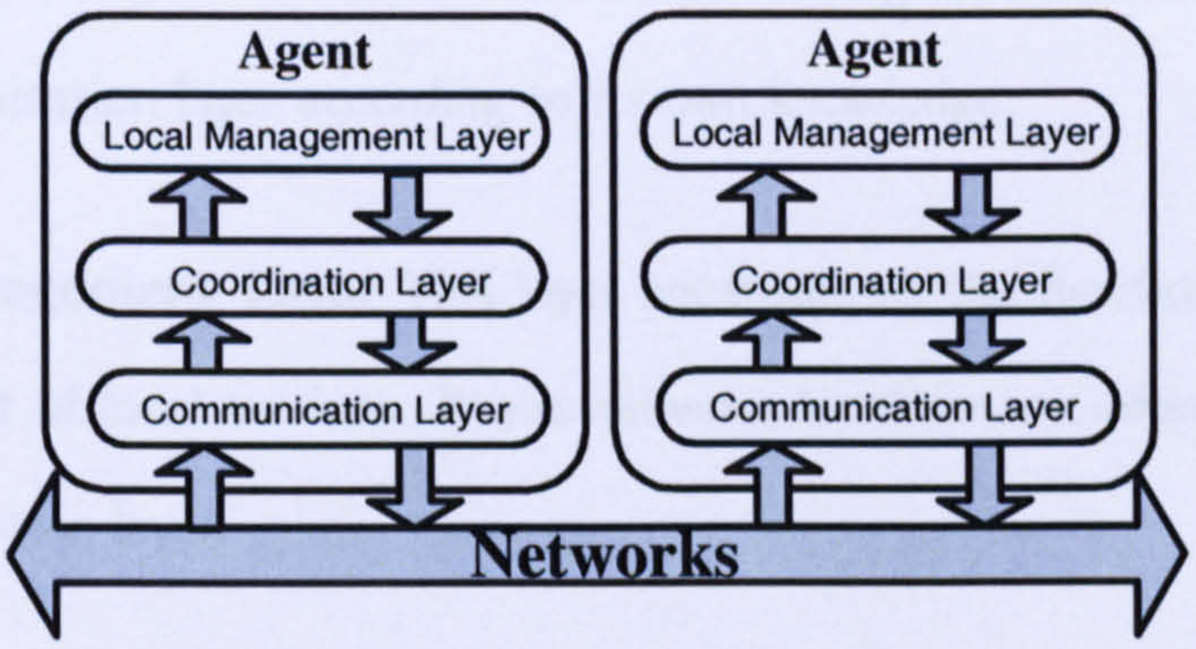

Figure 4.1: The basic structure of an agent consists of three interacting layers. For more details see [12].

4.2 Grid Resource Management Systems

Described in this section are software developed with the High Performance Systems Group in the Department of Computer Science at the University of Warwick. References to these software are made at various points in this thesis.

4.2.1 The A4 Agent System

The *Agile Architecture and Autonomous Agent* system (A4) [11, 12] addresses the general problem of resource management using an agent-based implementation where agents cooperate to discover available resources. This process is termed *service advertisement and discovery*. Every agent has knowledge about its neighbouring agents which process one another's service advertisement and discovery requests[13].

In A4, a hierarchy of agents is used to provide wide-area resource sharing. The agents are homogeneous and consist of a number of functional layers as illustrated in Figure 4.1.

- **Communication Layer** Agents use this layer to communicate with one another using common data models and communication protocols. An Agent Communication Language (ACL) can also be used by agents to exchange knowledge with one another.

- **Coordination Layer** This layer decides how the agent should act on the data at the communication layer according to its own knowledge.

- **Local Management Layer** This layer encapsulates the functions needed for the management of local services. It also provides local service information needed by the coordination layer.

This wide-area agent system has been integrated with a local-area Grid task scheduler known as *Titan*.

4.2.2 Titan Local Resource Manager

Titan [104] is a local-area workload management system used to select suitable resources for a particular task, given a varied, dynamic resource pool. The search space for the multi-parameter scheduling problem is large and not fully defined until runtime. Consequently, a *just-in-time* approach to performance prediction is adopted so that runtime variables and resource load can be used to assist task and resource allocation while maintaining prescribed service contracts.

An iterative, heuristic algorithm forms the basis of each local scheduler. This algorithm aims to minimise makespan, which is the latest completion time for a task, as well as processor idle time. The algorithm is written in such a way as to allow changes to be absorbed; these include the addition or deletion of tasks, and changes in physical resources including the number of hosts or processors. The approach used by Titan is to generate a set of schedules and to evaluate the schedules to obtain a measure of fitness. It then selects the most appropriate and combines them using operators (crossover and mutation) to formulate a new set of solutions. This process is repeated, resulting in a *fittest* solution. An important aspect of the algorithm is the use of predictive performance data from the PACE toolkit that forms the basis for its scheduling decisions.

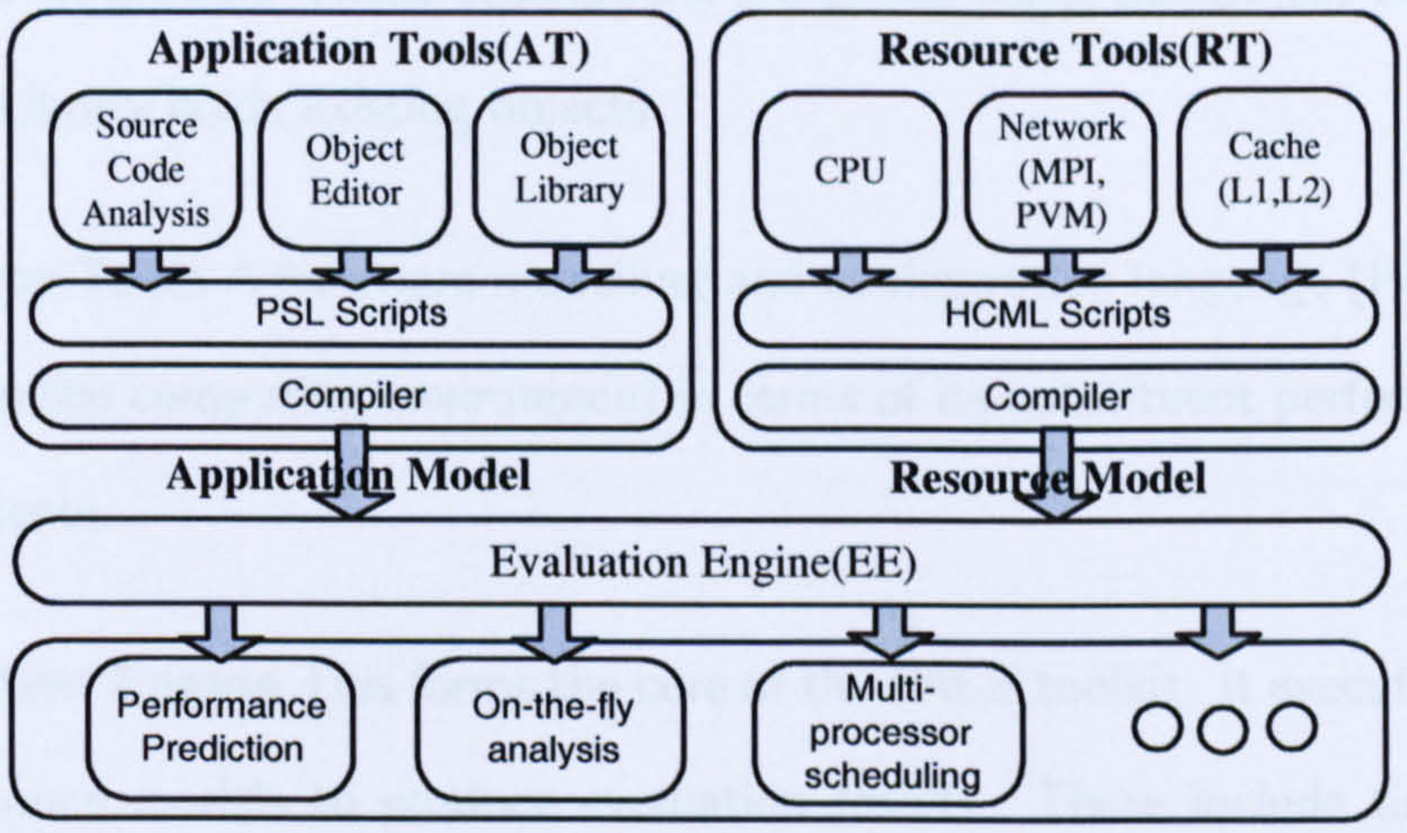

Figure 4.2: Components of the PACE Toolkit.

4.2.3 Performance Prediction with PACE

The *Performance Analysis and Characterisation Environment* (PACE) is an important component of the agent-based system because of its performance prediction capabilities.

PACE is a dynamic performance prediction modelling toolset used by high performance distributed applications. Some of the tools that it contains are for model definition, model creation, evaluation and performance analysis. It uses associative objects organised in a layered framework as a basis for representing each of a system's components. Moreover, the dynamic instrumentation technique used by PACE allows the automatic analysis of applications and hence, the production of reliable prediction results. These results are then used to steer the execution of these applications [1] dynamically. PACE also encompasses the performance aspects of application software, its resource use and mapping, and the performance characteristics of hardware systems [2]. The main components of PACE, shown in Figure 4.2 are:

- **Application Tools** The performance characteristics of an application and its parallelisation are described using the toolkit's performance specification language (PSL). The Source Code Analyser converts sequential source code components into perfor-

mance descriptions. These descriptions are edited using the Object Editor and the Object Library holds existing objects.

- **Resource Tools** A hardware modelling and configuration language (HMCL) is used to define the computing environment in terms of its constituent performance model components.

- **Evaluation Engine** This forms the core of the PACE toolkit. It executes completed performance models to produce evaluation results. These include time estimates and trace information relating to the expected application behaviour.

PACE is also used for dynamic multi-processor scheduling and for efficient resource management. Furthermore, it provides realistic performance predictions of expected application execution. The PACE toolset is comprehensive in its approach and is used in many different application areas [61].

PACE is based on a layered characterisation methodology and is an analytical-based approach. It also supports the entire software lifecycle including development, execution and post-mortem performance analysis [62].

4.2.4 Grid Resource Information

Dynamic information including CPU utilisation and network overhead, can be useful for the on-the-fly creation of the PACE resource models. Traditionally, these models are produced by running benchmark programs on different platforms. Additionally, it is desirable to add metrics to the application execution time, including the current memory usage and details of the execution environment, to the evaluation of the cost model for a request.

Furthermore, it would be beneficial for Titan to share its local resource information to users in other administrative domains, in the Grid environment so that its resources can be used efficiently. By using standard Grid protocols, it can become interoperable with

other local resource managers and user applications.

Service advertisement in A4 is carried with nearby neighbouring agents, namely its upper and lower agents. Consequently, the propagation of resource information to a wider infrastructure occurs only after many iterations of service advertisement and discovery. This process may take a long period of time and result in considerable resource information duplication. Therefore, it would be useful to have a framework providing soft state information.

The Globus MDS fits in well with the above systems as it provides resource information from multiple domains, using standard interafces.

One issue arising from the Grid being a large number of networked resources, is resource contention, which itself will also affect application performance. It is therefore important that the effects of resource contention are carefully managed, for example through resource monitoring and scheduling. The material for the following section, supported in [66], presents a model where the Grid Information Services can be used with an agent-based resource management system [11] to discover and monitor resources within large-scale Grid systems.

4.3 Grid Information Management using Software Agents

Although a local scheduler has information about the resources in its local environment, it has no knowledge about resources in other local scheduling environments or in other administrative domains. As broker agents discover resources or receive service advertisements from neighbouring agents, they have to store this information. While each agent has enough information to propagate a task to its 'best suited' neighbour, there is no 'global' information repository which agents can access on demand. Service advertisement in the agent hierarchy is currently only carried out with nearby, neighbouring agents. Therefore, an agent advertises its service information to its upper or lower agent only. In this case,

resource information is propagated to a wider infrastructure only after many iterations of service advertisement and discovery, which can take a long time. There is also the danger of a large amount of resource information duplication. The use of the MDS will overcome these problems.

The movement of service discovery requests from one agent to another will occur in the same way within a virtual organisation or across virtual organisations. This method therefore allows the agent hierarchy to be scalable.

The model shows how agents implement the GRIP and GRRP protocols to pull information from resources and push information into aggregate directories. It also demonstrates how such agents can implement an automatic referral mechanism to discover other aggregate directories and hence resources in other virtual organisations.

4.3.1 Architecture of Proposed Model

Within the Grid environment, each resource under local scheduler management runs a local GRIS. A GRIS can service requests for specific resources, but in this model, a GRIS is configured to register itself with an aggregate directory service (GIIS) via the local scheduler, so that information can be passed onto other agents.

A study of the systems above indicate that the A4 agents and the MDS architecture share two major features: a hierarchy and an information storage capability. Further work demonstrates that the agents' structure can be mapped onto the MDS, resulting in the integration of both architectures. The following describes how this mapping and integration are done.

Figure 4.3 shows an overview of the hierarchy of agent-based Grid information services components. Each agent interfaces with a GIIS which has information about all the resources in its administrative domain, including all its local resource managers. The *scope* of information for each agent spans downwards towards all the other agents which

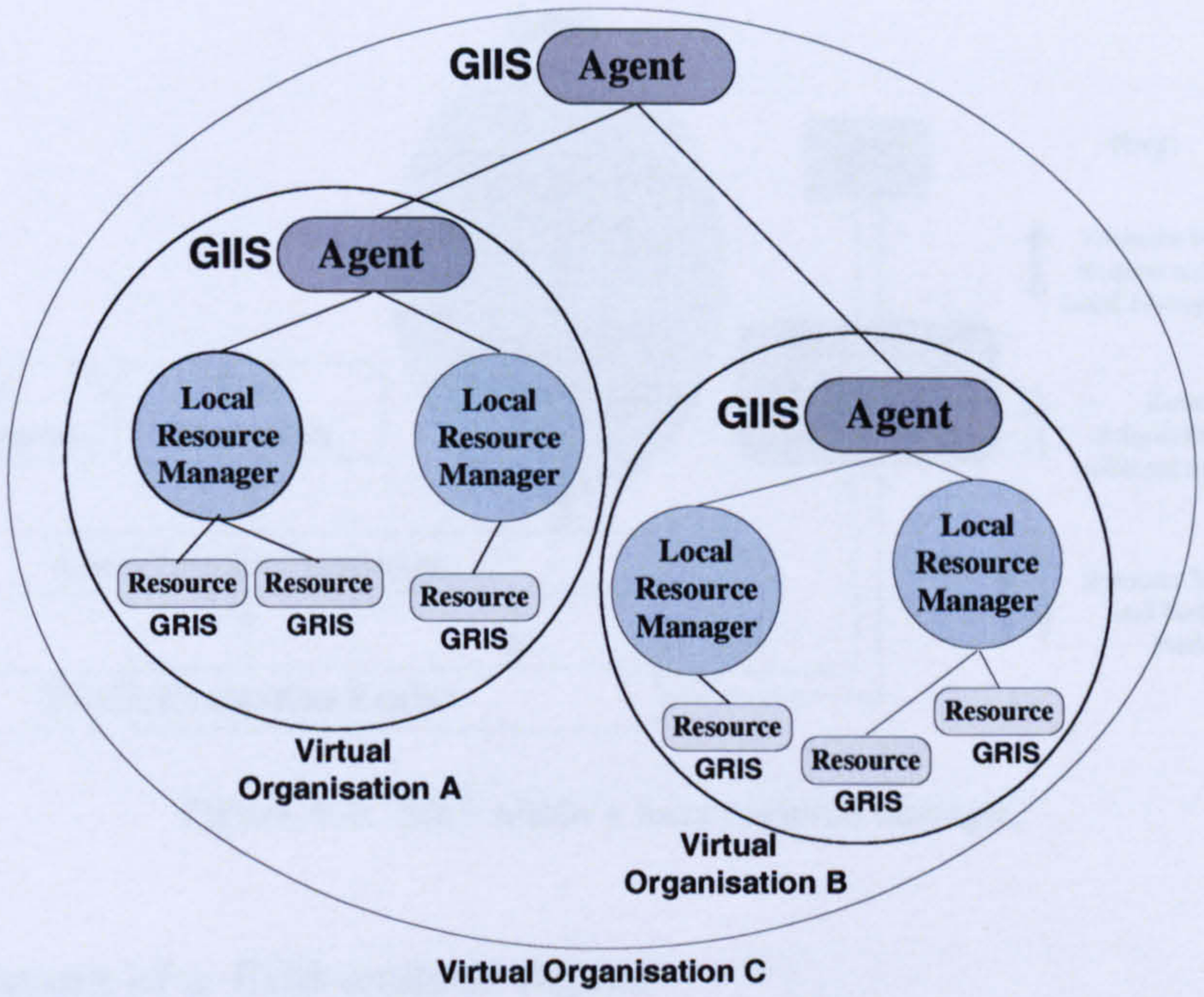

Figure 4.3: Information services structure with a hierarchy of virtual organisations.

have registered their resources with the former agent.

Therefore, homogeneous agents which communicate with one another, are arranged in a tree-like hierarchy. To be able to act as a high-level broker to the underlying metasystem and to carry out performance prediction, those agents need to have accurate resource information. The Globus MDS provides the information required for that purpose. Every agent will be closely integrated with its own GIIS which will contain information about all the resources within the VO.

Figure 4.4 shows a more detailed diagram of the components of the local resource manager. The latter pulls information from its information providers via its *Resource Monitoring* component and pushes that information to the agent's GIIS via its *Resource Information Reporting* module. The GRIS component is made up of the information providers and the *Resource Monitoring* module.

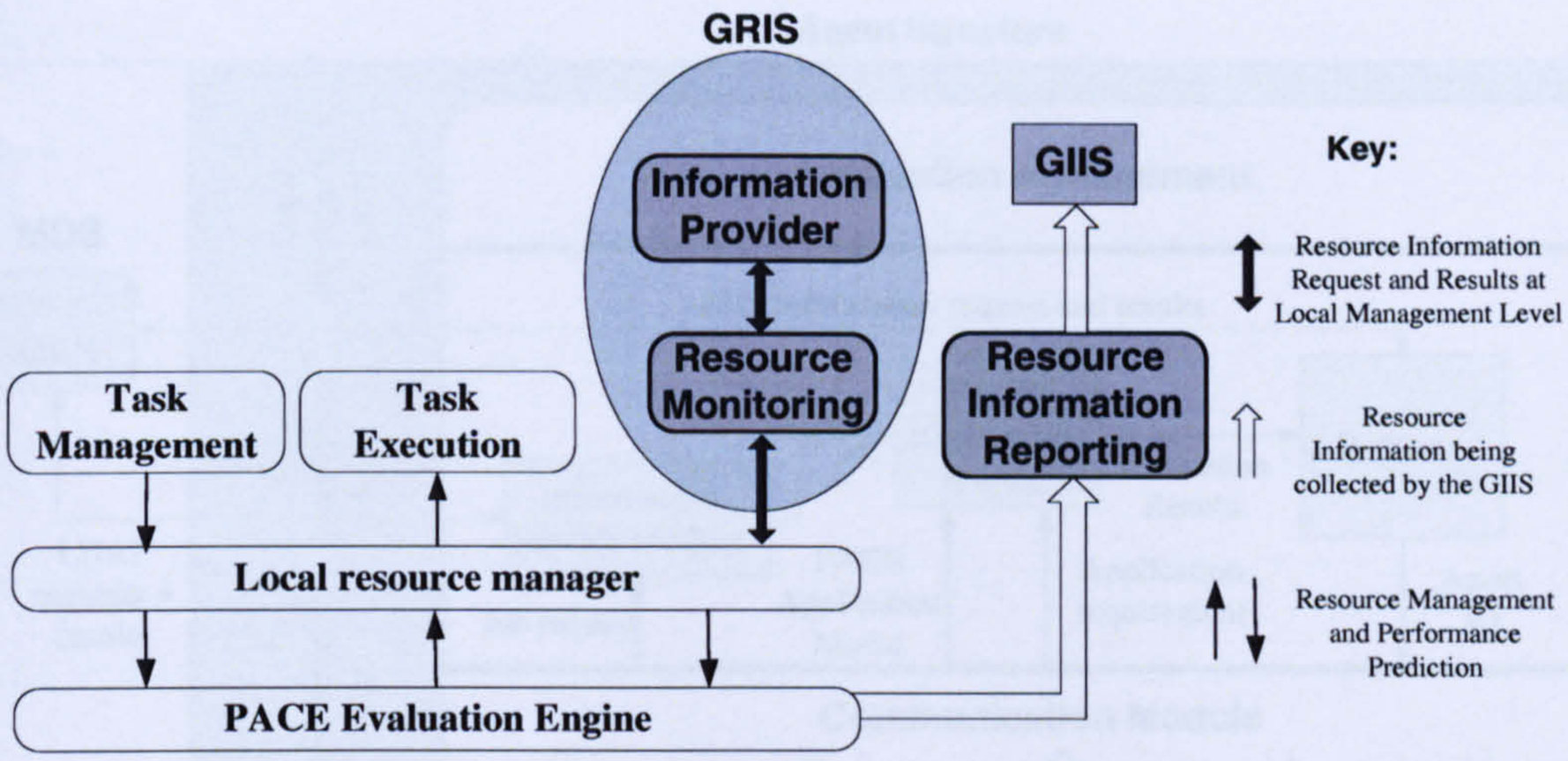

Figure 4.4: GRIS within a local resource manager.

4.3.2 Structure of a Grid-enabled Agent

An agent will not only act as a broker to the underlying local resource manager but it will also interface with a Grid Information Service. The agents can therefore, automatically refer to other neighbouring agents in order to access the whole set of their resource information, if their own resources do not meet users' job requirements. The overall system provides an autonomous Grid Information Service which can locate resources across virtual organisations. The way in which the A4 software agents will interface with the MDS is described below.

An agent's structure is made up of three layers: the *communication*, *coordination* and *local management* layers. The coordination layer further splits into the following components: the Information Service Module, the PACE Evaluation Engine and the Referrer. The structure of an agent and the interface to its GIIS are shown in Figure 4.5. The various functions of these components are now described.

One of the functions of the Information Service Module (ISM) is to convert a service discovery request in XML format to an LDAP-based request. This request is serviced by the GIIS which interfaces with an agent. The GIIS contains resource information about all the local resource managers in the corresponding virtual organisation. It also has potential

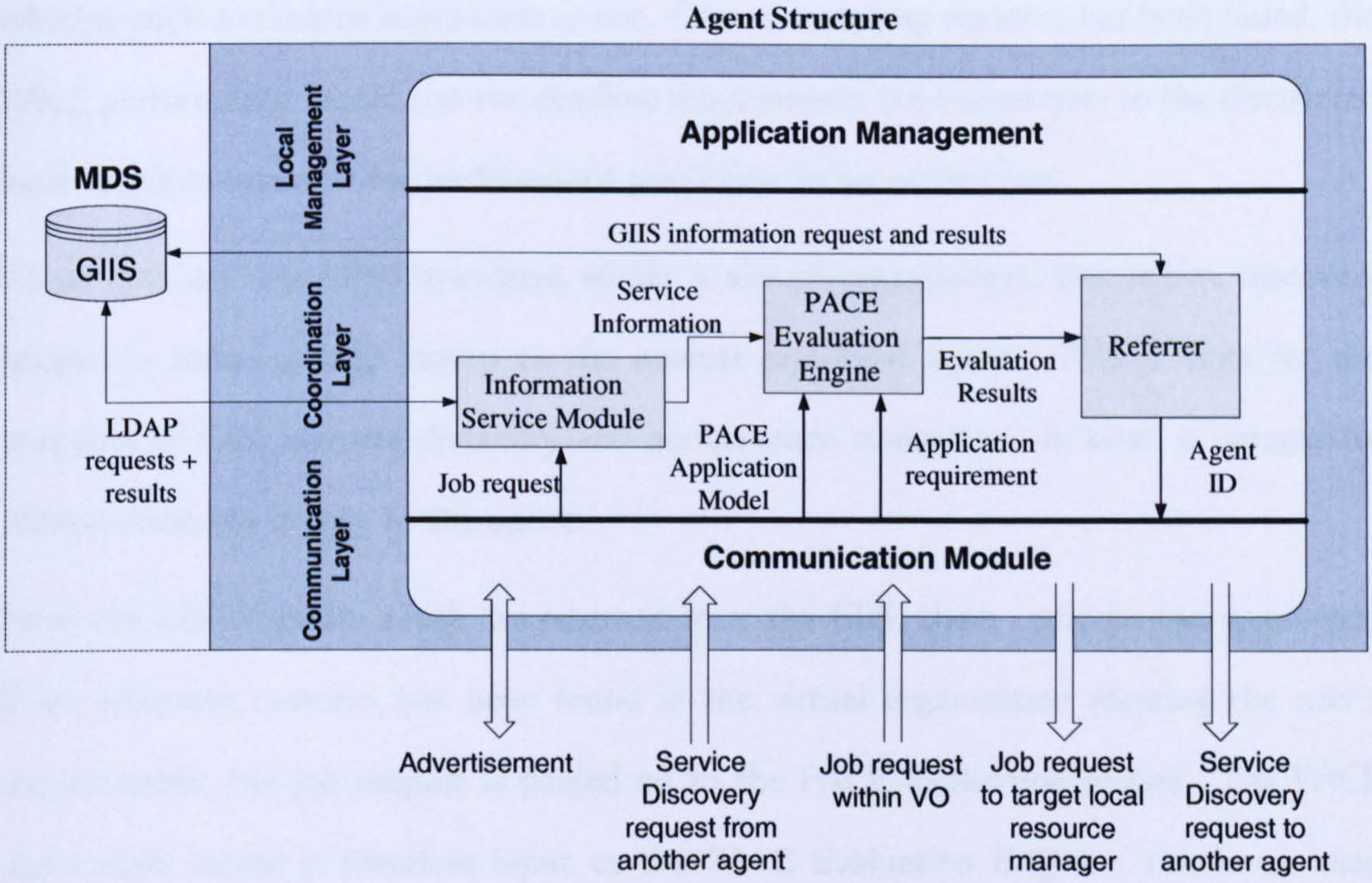

Figure 4.5: The agent-based GIIS structure where the agent interacting with its GIIS is shown. The information flow during resource discovery and performance prediction are indicated by the respective arrows.

access to information about other virtual organisations' resources since it receives service discovery messages. Therefore, the aggregate directory structure is accessed by the ISM when a new service discovery request arrives. The purpose is to determine whether there is a local resource manager with available resources which can process the job whilst satisfying the user's requirements, in terms of the hardware specifications and the period of execution. In short, querying the GIIS is done via the LDAP request that has been generated beforehand.

A user wishing to run an application on the Grid, submits the actual application and a *job request*. The latter consists of resource requirements, a PACE application performance model and performance metrics which include deadline requirements. These deadline requirements are obtained from previous results by using the PACE Evaluation Engine. On receipt of the job request, a *service discovery request* is generated, containing resource requirements. The corresponding virtual organisation GIIS is then contacted to discover

whether such a resource is available or not. Once a matching resource has been found, the PACE performance model and the deadline requirements are copied over to the discovered local resource manager for performance prediction to be carried out.

When jobs are submitted anywhere within a virtual organisation, the service discovery request is automatically moved to the nearest high-level agent. This is done for the purposes of Grid resource discovery and performance evaluation. In brief, a *request for service discovery* is sent to the agent.

From the LDAP results which are returned from the GIIS, there could be two outcomes. If an adequate resource has been found in the virtual organisation meeting the user's requirements, the job request is passed on to the PACE Evaluation engine. The PACE application model is therefore input to the PACE Evaluation Engine. The latter then performs performance prediction and passes the evaluation results to the referrer. Depending on the evaluation results, the referrer could undertake either of the following. If a resource has been found which can meet the job's predicted execution time and fits onto an existing schedule, it gets scheduled. If this is not possible, the referrer evaluates which agent to contact next for GIIS information; this agent could be an upper or lower agent. Subsequently, the service discovery request is moved to that agent and the above process starts all over again. Likewise, if no adequate resource has been found in the GIIS from the outset, the referrer determines an agent to which the service discovery request is forwarded.

The agent receives both service advertisement and discovery messages via its communication module. It interprets the contents of each message and submits the information to corresponding modules in the coordination layer of the agent. For example, an advertisement message from another agent will be handled by the Information Service module which will consequently add the new information in its own GIIS. Moreover, the communication module is responsible for communicating service advertisement and discovery

messages with other agents.

4.3.3 Information in the Agent-Enhanced GIIS

Cooperation amongst homogeneous agents is defined by the service advertisement and discovery taking place for the purpose of resource management. An agent has access to the following information stored in the GIIS:

1. **Resource information** in the VO.

2. **Agent ID** This is the contact information for an agent. The agent ID is very useful in contacting other agents to search their GIIS. Each agent will initially store its upper agent ID and later when other agents register with it, it will store lower agent IDs.

3. **Service Information** This is where performance-related information about resources is stored. The agent uses this information to evaluate the performance of resources and to ensure that the performance metrics are satisfied. The service information is also used as part of service discovery decisions.

The GIIS will thus contain resource information, as described earlier, as well as performance information.

A resource's GRIS can be directly contacted for information. However, in a Grid environment with a large number of heterogeneous resources, it is more helpful for the local resource manager to contact its agent to look up its GIIS. Each agent will have one or more local resource managers, with each resource manager monitoring one or more resources.

The agent ID is used as follows. The resource needed by a job request might be found in the local virtual organisation, but if no such resource is found, the agent will use one of its agent IDs to contact another agent via its communication layer. More specifically,

at the agent's coordination layer, the GIIS is accessed and the ID of the selected agent to contact next is retrieved. The service request is then sent to that agent. It might be appropriate to contact the agents in the hierarchy in a breadth-first search fashion. However, no method of traversing the agent hierarchy is particularly preferred. If a suitable resource and schedule have been found in the GIIS of one of the agents in the hierarchy, the corresponding local resource manager is contacted and the job is forwarded to it. Consequently, at each stage until a schedule on a suitable resource is found, the job remains on the submission host, but the job request moves from one agent to the next.

It is intended that an LDAP data model is used to represent all the information needed to carry out resource discovery and monitoring, and performance prediction. In short, the information is classified as follows:

- All of the resource information (static and dynamic) within the VO;
- Service information pertaining to the performance of resources in the VO (and others);
- ID of other agents which an agent knows about, including its lower and upper agents.

4.3.4 Information Flow Overview

There are three main types of cross-VO communication between agents:

1. Service advertisement and service discovery messages;
2. Service discovery requests moving from one agent to another;
3. Jobs moving from the portal to the target local resource manager (the A4 portal allows jobs to be submitted to Titan);
4. Job requests moving from one agent to another.

When a new virtual organisation wishes to join the Grid, its agent advertises its service information to other agents. At the same time, other agents are in the process of discovering new agents. Once the new agent has chosen its upper agent, it registers its GIIS with it and the new virtual organisation is now part of the Grid. The performance offered by resources is likely to vary over time, and if a virtual organisation wishes to cease to be part of the Grid, its agent should unregister from its upper agent, thus its resources are no longer available to execute jobs. In this way, the rest of the Grid is unaffected by the removal of a GIIS.

Each resource has an upper local resource manager which continuously monitors its performance. The existing GIIS schema has to be extended from having only hardware information to including the agent ID and service information as well. This is necessary because when a resource has been found, the job request is not sent to that resource directly but to its local resource manager. This method thus allows the resource manager to perform performance prediction before actually submitting the job to the targeted resource.

4.3.5 Service Advertisement and Discovery

When a virtual organisation wishes to register with another high-level one, the agent uses the GRRP protocol. LDAP queries can be sent from the higher-level GIIS to the lower level one, but there is no existing mechanism for the lower-level GIIS to search higher-level GIIS with which it registered.

This drawback can be avoided by using the agent service advertisement mechanism at the time the virtual organisation is registered. Therefore, the new virtual organisation's agent advertises its service information to other agents. This happens with its upper agent in the first instance. Once the new agent has registered with its upper agent, a record of the latter's ID is stored in the new agent's GIIS.

Service Discovery occurs when other agents' GIIS need to be queried for resource information. The lower or upper agents are thus contacted.

4.4 Grid Information Services Supporting Resource Management

It is the responsibility of local resource managers, for example, within cluster resources, to make decisions about the specific resources on which jobs are processed, depending on load and availability. Therefore, information about the structure and the state of schedulers should be provided to Grid brokers, which will then decide where to schedule the applications.

This section, which is also found in [72], illustrates one way in which low-level scheduling information is collated from multiple sources and is incorporated into the unified Grid information view. The existence and availability of Grid resource information also allows reliable application performance prediction to be carried out using the Warwick PACE (Performance Analysis and Characterisation Environment) system.

4.4.1 Local Scheduler as an Information Provider

The process of scheduling scientific applications to be run on the Grid involves many steps, one of which is to obtain exact state information from local schedulers. The scheduler used in this section is *Titan* [104] which has been introduced in Chapter 2. Each instance of the scheduler manages a resource pool where the characteristics of the set of schedules and those of the resources are highly dynamic. In a Grid environment, it must be possible for this kind of cluster information to be propagated and made available to other remote administrative domains. Only then can the superscheduling of scientific applications take place, based on the local scheduling information of multiple sources.

In this context, an information provider is a service that provides a subset of useful information about resources participating in the Grid. Moreover, the structure of the MDS

offers a unified solution to the distributed nature and fail-prone information providers. There is a need for information services to be as distributed and decentralised as possible, with providers located on or near the entities they describe [21]. Therefore, it is reasonable to have the scheduler act as an information provider or to have a database near the scheduler providing such services. Additionally, having the Titan scheduler as an information provider increases the likelihood of obtaining dynamic and reliable information about available resources. Likewise, the role of the Grid information service is to focus only on the efficient delivery of state information from a single source, that is the particular information provider. Furthermore, information providers are independent of one another when registration messages are sent from a GRIS to a GIIS. Therefore, no information provider should prevent information from being obtained about other components of the system, resulting in a robust system even in the face of failure.

One of the ways in which scheduler information can be made available to the MDS is by speculative evaluation [41] where information from the scheduler is generated at a regular interval. This information is placed in a local back-end database which the GRIS can access upon request, as shown in Figure 4.6. This method has been implemented on the Grid testbed at the University of Warwick and it has both advantages and disadvantages. The benefits are that the scheduler itself is not overloaded, since a central repository is accessed for the values of scheduling attributes. Moreover, if the scheduler fails, the latest values of scheduling attributes are still accessible. These values have a time-to-live attribute attached to them, specifying the length of time for which the values are valid. On the other hand, the data in the back-end database is very dynamic and hence, the scheduler might have a very high write frequency and a comparatively low read frequency. The data is also written on a frequent basis irrespective of the fact whether it is read. However, it is found that since the database is local to the scheduler on the back-end of the GRIS server, this does not affect the way its information is pulled from the aggregate directory.

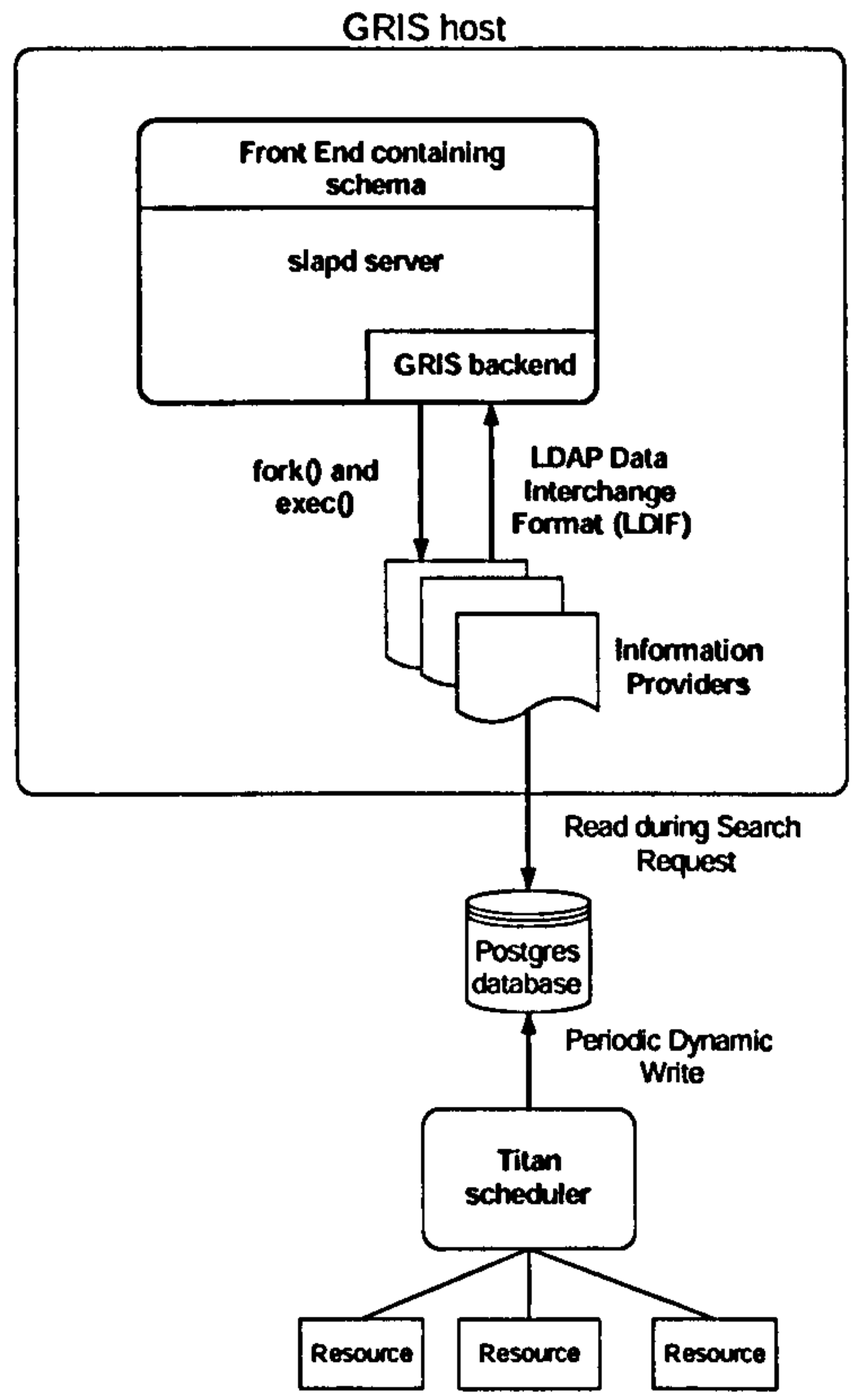

Figure 4.6: Reading and writing dynamic scheduling information.

The back-end database is relational (Postgres) [92], allowing read and write transactions to be handled efficiently. A number of information providers are then created using JDBC (Java Database Connectivity) [93] to access particular fields in the database, corresponding to specific scheduling attributes. It is thus possible for a GRIS to access the current value of any scheduling attribute through an information provider which accesses the Postgres database via a JDBC driver. The LDAP schemas are modified to allow the new attributes to be readable and the scheduling attributes' distinguished names are automatically reconfigured when viewed from higher GIISes. The result is a hierarchy of virtual organisations sharing resource information.

Other Alternatives for the Information Providers

A slight variant on the above implementation was to use an LDBM (LDAP Database Manager) database [88] instead of the relational one. This allows the GRIS to pull dynamic information from information providers which access this LDBM database in a speculative evaluation method. The information providers for the scheduler are then written using JNDI (Java Naming and Directory Interface). In this case, the method used to access a GIIS would be similar to that retrieving information from the scheduler. But instead of transactions, the scheduler would use commands of this type:

```
ldapmodify -x -h lab-68.cslab -D "Mds-Software-deployment=Titan
scheduler, Mds-Vo-name=local,o=grid" -f modify-makespan.ldif
```

The advantage of using an LDBM database back-end for the scheduler is that there is no conversion from LDIF to other data formats, thus keeping the data model uniform. Furthermore, both reads and writes are allowed on such a database, unlike the shell-type backend server used by the GRIS.

Yet another method which could be used on its own or concurrently with the speculative evaluation method above, is the eager evaluation. This method focuses on caching data which is generated when a search request is first received. Therefore, the scheduler information which the GRIS has accessed from its back-end database can be stored in cache for a configurable amount of time. There are a number of advantages with this method: the load on the GRIS host is reduced and the time taken to service a search request is less. However, the drawback lies in the relative staleness of the information. At the other end of the spectrum, there is lazy evaluation where the scheduler generates information only when a search request is received by the GRIS. This method provides the most up-to-date information but the service time increases as well. Another cost is the increased load on the GRIS host, which each service request carries.

To obtain dynamic information from the scheduler, the speculative evaluation method is the most appropriate. Up-to-date dynamic information is required, hence making a purely eager evaluation infeasible; on the other hand, a lazy evaluation will only increase the load on the GRIS host. Consequently, a speculative evaluation method is judged the most appropriate in this case.

4.4.2 Implementing the Titan Scheduler as an Information Provider

This sub-section details how information is pulled by a GRIS from the Titan scheduler for attributes including makespan [104], queue length and whether the genetic algorithm is used. The MDS version used in this work is MDS2.4. While the MDS is installed on a node called `frog.dcs.warwick.ac.uk`, Titan is running on another node in the same LAN, `soda.dcs.warwick.ac.uk`. Before any information from Titan is integrated into the MDS, a search on the GRIS returns output from the core information providers alone. The full output is given in Appendix D.1.

The new Titan scheduler information providers on `frog`, are grouped in a directory called

/home/user/gram_reporter. Details of these custom information providers are given in Appendix D.2.

Moreover, the following is added to the $GLOBUS_LOCATION/etc/grid-info-resource-ldif.conf file, for each of the scheduling attributes required:

```
##########################################
# GRAM Reporter for Titan Information #
##########################################

# Titan GRAM reporter - Phenotype
dn: Mds-Software-Component=Phenotype, Mds-Software-deployment=Titan scheduler,
    Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/user/gram_reporter
base: access_phenotype_infoprov.pl
cachetime: 0
timelimit: 20
sizelimit: 20

# Titan GRAM reporter - Deadline
dn: Mds-Software-Component=Deadline, Mds-Software-deployment=Titan scheduler,
    Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/user/gram_reporter
base: access_deadline_infoprov.pl
cachetime: 0
timelimit: 20
sizelimit: 20

# Titan GRAM reporter - Dominant type
```

```
dn: Mds-Software-Component=Dominant type, Mds-Software-deployment=Titan scheduler,
    Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/user/gram_reporter
base: access_dominanttype_infoprov.pl
cachetime: 0
timelimit: 20
sizelimit: 20

# Titan GRAM reporter - Iterations
dn: Mds-Software-Component=Iterations, Mds-Software-deployment=Titan scheduler,
    Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/user/gram_reporter
base: access_iterations_infoprov.pl
cachetime: 0
timelimit: 20
sizelimit: 20

# Titan GRAM reporter - Dominance
dn: Mds-Software-Component=Dominance, Mds-Software-deployment=Titan scheduler,
    Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/user/gram_reporter
base: access_dominance_infoprov.pl
cachetime: 0
timelimit: 20
sizelimit: 20
```

A new schema is also created so that both the GRIS and GIIS can understand the new scheduling information. The new schema is called `warwick-scheduler.schema` and is given in Appendix D.3. Additionally, a reference to that schema file is required in the `grid-info-slapd.conf` file as shown in Appendix D.4.

Consequently, after the MDS server is stopped and restarted, and `grid-info-search` launched, output from both the core and Titan information providers are obtained, as shown in Appendix D.5. Various filters can also be applied to the search command.

4.5 Performance Evaluation of MDS2

As Grid applications are likely to be influenced by dynamic middleware, it is crucial that they experience a reliable performance for the proper utilisation of resources and for accountability purposes. Consequently, the approach taken in this section is to investigate and evaluate a level of performance obtainable from the MDS. A better understanding of the performance of the GRIS (Grid Resource Information Service) and subsequently the GIIS (Grid Index Information Service), leads to a more standard way of formalising the performance of a Grid Information Service. This is achieved by defining a number of performance metrics which are analysed against a set of different information gathering methods and GRIS back-end implementations.

The contribution of this section of this thesis is therefore an analysis of the performance obtainable from the GRIS and the effect on Grid applications. More specifically, the way in which data is provided by information providers to the GRIS, is studied. The latter advertises Grid resource information about a node, collated from multiple information sources. Depending on whether caching is enabled in the GRIS, it is possible that all the information providers are executed for a query. Alternatively, a periodic information update mechanism can be implemented. These various methods affect the query performance in different ways; therefore in this section, a GRIS will be repeatedly queried under specific

conditions, and the performance obtained will be analysed. This work is supported in [69, 67].

4.5.1 MDS Evaluation Methods

There are a number of ways by which an information provider can supply information to a GRIS. The definitions of the meanings of the different evaluation methods [41] are:

1. *Lazy evaluation*: Obtain freshly generated information on receiving a search request.

2. *Eager evaluation*: Obtain freshly generated information on receiving the first search request, and cache it in the GRIS. Thereafter, check the cache to see if the subsequent search requests can be serviced. If the cache TTL (time-to-live) has not been reached, then the requests are serviced out of the cache. Otherwise, obtain and cache freshly generated information (lazy evaluation).

3. *Speculative evaluation*: Information is generated frequently and placed in a recognised location. On receiving a search request, service is provided from information in that location. There is no caching in the GRIS in this method. Here, the information being returned for the search request may not be fresh as in the lazy evaluation method, but it is readily available and is frequently updated.

4.5.2 MDS and its Performance

Overview of the Performance of the MDS

Performance is an issue for Grid applications as they execute on heterogeneous resources with, for instance, variable bandwidths and latencies [6], changeable processor speeds and memory availability. Since it is difficult to characterise the performance of such a dynamic and heterogeneous environment, it is increasingly important to provide a reliable performance through quality of service.

The performance of a query to the MDS cannot be predicted with a formula. This is because the predictability of the performance decreases with the complexity of the GIIS hierarchy. Moreover, the length of time a query might take to answer depends on the time-to-live (TTL) data [30]. The approach normally taken to counter this performance variability is to ascertain that the data being requested is in cache. When queries are sent to the MDS, they will be serviced by data in the cache which is regularly refreshed. Another method is to increase the TTL value for the data, though this might not be feasible for dynamic data. These techniques could be applied to any GRIS or GIIS independently or at the same time.

Due to the inconsistencies above, it is crucial to perform tests at the smallest unit that can exist in the MDS hierarchy, namely the GRIS. Understanding how information is provided at the lowest level is important because performance is unpredictable and depends on the complexity of the hierarchy. Furthermore, the number of information providers is large, and as they are the original sources of information, they are accessed frequently. These factors affect the performance with which this information is propagated upwards to the higher level GIIS nodes.

Assessment of Performance

Middleware has a number of characteristics: size, cost, complexity, flexibility and performance. The importance of each characteristic depends on the application. One way of assessing performance is to measure the rate at which requests can be sent and results received through a given system. The most performant middleware system is the one with the least time taken for the messages to pass through it. When time measurements are taken from the sending moment to the receiving moment, no application-specific setup is considered. The measurement thus effectively represents the behaviour of the middleware and the network and operating system below it.

The difference must be made between *high performance* and *high quality* middleware. Middleware performance can be measured by the time taken to carry out an operation. However, the fastest middleware is not the only issue for many middleware applications. Other issues including scalability, flexibility and adaptability, ease of use, tool support, and standards conformance could be more important, depending on the application. Nevertheless, these characteristics are very difficult to measure since they are relatively subjective. Furthermore, scalability depends on performance; for example, for an information service to handle a large number of concurrent requests, it needs to process each request at a high speed. If the information service takes a shorter time to process one request, it can handle more requests in a particular period of time.

However, high performance alone does not guarantee high scalability. For instance, it is possible that some information services perform well when run single-threaded but as soon as multiple threads of control are used, the performance greatly degrades. This occurs because all the threads are competing for the same resources and so must block and waste cycles, attempting to acquire the locks required to use the resources.

In the experiments described in this section, reliable performance is built on an analysis of the performance of the Grid Information Service. Since it is important to analyse the GRIS, this section of the thesis investigates the different information provider mechanisms and the benefits of using different caching strategies.

4.5.3 Experimental Environment

The experiments were carried out on a Grid testbed at the University of Warwick and were based on MDS 2.1. This particular version of the MDS was chosen because at the time of writing, MDS 2.x was being utilised in the majority of UK e-Science projects [116] and US testbeds including NASA's Information Power Grid [60]. Across the various experiments, the following agent setup was maintained. Agents make request queries to the MDS,

which are sent from a set of ten machines (*mscs-02* to *mscs-11*). With a maximum of 500 agents simultaneously making queries over a period of 10 min, the desired effect was to load-balance the queries and to sustain the MDS querying. The maximum number of agents attributed to one machine is therefore 50.[1] The time it takes for each request to be serviced is measured and an average response time is calculated. Moreover, every agent sleeps for 1 s before sending the next request.

To test the scalability of the MDS, a GRIS was set up.[2] For the lazy evaluation experiment, the MDS default cache time values were used. Furthermore, for the experiments, the core information providers included by default in the MDS were used. These include the GRAM reporter which provides information about the *fork* job manager. The aim of the experiments was to analyse the performance of the GRIS with a minimal number of information providers which offer relevant status information about the Grid.

To evaluate the performance of the GRIS, an agent application was developed to query either a GRIS or a GIIS. One set of experiments was carried out with the agent using the Java CoG Kit [119] libraries and the other set with the Globus C APIs. The aim was to determine whether the implementation language actually affected the performance obtained when querying the MDS. The Globus C and Java CoG APIs have been used instead of visualisation tools because it is typical for Grid applications to use those APIs directly.

Figure 4.7 shows the different components of the experimental setup with a focus on speculative evaluation (SE).

---

[1]The *mscs* machines each has the Linux operating system with kernel 2.4.18-27.7.x, a 2.4 GHz processor and 512 MB RAM. The *mscs* machines are also on an Ethernet LAN and they are connected to the GRIS host by a 100 Mb link.

[2]It was set up on a Linux kernel 2.4.18-14 machine (M_1) which has a 1.9 GHz processor and 512 MB RAM.

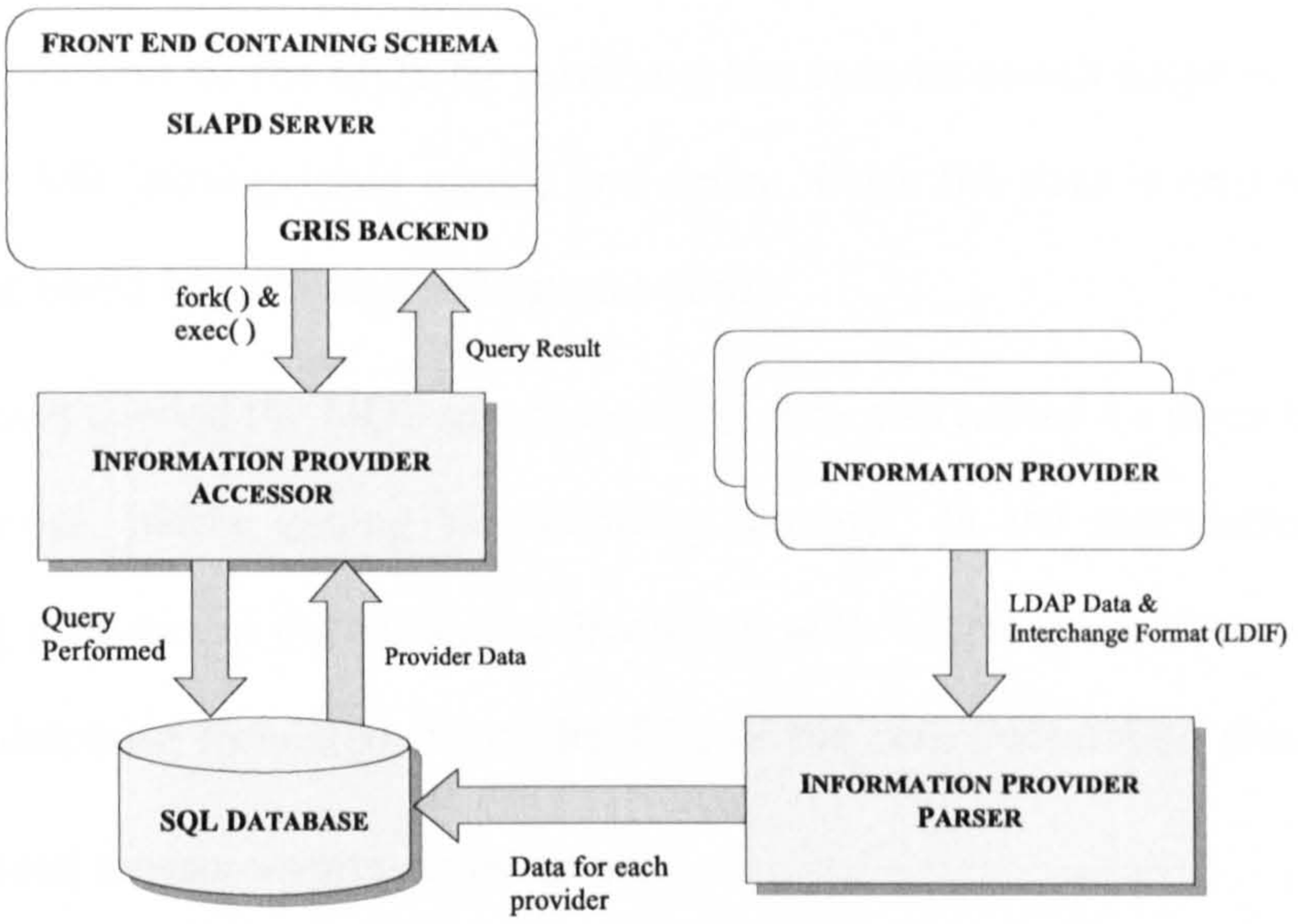

Figure 4.7: Components of the system.

Experimental Factors

The approach in this section is to consider that performance is defined by a number of factors, including response time, throughput, the total number of responses and system load. The scalability of the MDS directly depends on the performance observed.

The average response time in seconds, is the mean time it took each query made by an agent, to be serviced by the respective MDS component. The average throughput denotes the mean number of queries which are processed by the GRIS per second. Each experiment also keeps a count of the number of successful queries processed by the MDS. Moreover, the system load has been measured as the load on the system during the last minute (1 min load average) and during the last 5 min (5 min load average). The load average is a measure of the number of jobs waiting in the run queue.

In this section of the thesis, the complexity of queries is not assessed. Therefore, the complex relationships amongst data objects such as the sophisticated searches on objects,

are not being tested. The experiments only query the MDS for all the data available; Grid applications are likely to carry out such an operation to discover the status of the Grid. Thus, queries are sent to the MDS by specifying the *subtree* search scope which searches from the base DN (distinguished name) and down. Once the data is returned from the MDS, filtering could happen as a subsequent step.

Up to 500 agents queried the MDS simultaneously; they also waited 1 s when they received a query response, before issuing the following request. In the speculative evaluation method using a relational database, the frequency with which the information providers wrote to the database simulated the cache TTL of the core information providers.

The performance measurements obtained are:

- Average response time in seconds ($\mathcal{R}_T$). This is the average time from sending out a query and receiving the response, across all the successful queries. The theoretical maximum is the length of the experiment, which is 10 min.

- Average throughput in terms of the number of queries answered per second ($\mathcal{T}$). This is a server-side metric which demonstrates whether the MDS scales with an increasing number of concurrent queries.

- Total number of successful query responses ($\mathcal{R}_e$). A successful query is defined that occurring when MDS results return to the particular client, without timing out. This metric is a server-side one, showing the total number of queries serviced throughout the whole experiment duration.

- 1 min load average ($\mathcal{L}_1$). The load average is indicative of the MDS being under heavy usage, which increases the average response time and time-outs.

- 5 min load average ($\mathcal{L}_5$).

4.5.4 Experiment Contexts and Results

The behaviour of the MDS can only be understood after the collection of performance data for a long period of time. Therefore, the experiments were carried out five times and statistical analyses are carried out: average, standard deviation and confidence interval.

The experiment results graphs show the situations below.

GRIS Back-end Implementations

The results shown in the graphs in all experiments include the following set-ups, as given in Figure 4.8.

	Cache TTL	Information Provider Execution	Database	Information Provider Accessor
Lazy Evaluation (LE)	0	For each query	-	-
Eager Evaluation (EE)	≠ 0	On cache expiration	-	-
Java Speculative Evaluation (PostgreSQL)	0	Every minute	PostgreSQL	Java
Java Speculative Evaluation (MySQL)	0	Every minute	MySQL	Java
Perl Speculative Evaluation (PostgreSQL)	0	Every minute	PostgreSQL	Perl
Perl Speculative Evaluation (MySQL)	0	Every minute	MySQL	Perl

Figure 4.8: Features of the different GRIS back-end implementations.

In all of the speculative evaluation methods, the information providers themselves are written in Perl rather than Java due to their shorter execution time.

The experiments have also been repeated using C agents in order to determine whether the implementation language of the application querying the MDS makes a difference to the overall performance. It was found that there was no major difference between Java and C agents. Thus, Java agents will be used to show the results obtained for measuring only the performance of the MDS and not that of the applications.

Additionally, experiments have shown that there is no marked difference between the performance of an agent written in Java CoG and one written in JNDI (Java Naming and

Directory Interface). Therefore, all the experiments will involve Java CoG.

Experiment: GRIS scalability with Java CoG agents

This experiment tests the way in which the scalability of a GRIS changes with an increase in the number of Java CoG agents. Moreover, the six GRIS back-end implementations discussed previously have been implemented and the results obtained are shown in Figures 4.9 to 4.11.

From Figure 4.9 it is seen that the average response time generally increases with up to 150 concurrent agents but it slowly stabilises when that number of agents increases. It has been found that the best average response time consistently results from the EE implementation.

The lazy and Perl SE methods have a similar average response time with an increase in the number of concurrent agents. However, the Perl SE has a slightly better response time. The Java SE implementation produces the most significant increase in average response time with an increasing number of agents. This is due to the overhead caused by running the Java Virtual Machine (JVM) for each query. Furthermore, the Java SE methods using GRIS caching and both a PostgreSQL database and a MySQL database produce similar behaviour. This can be explained by the fact that on average, for each query, the data required is in cache and therefore, accessing the database does not affect the average response time from the GRIS. Nevertheless, it can be seen that once the GRIS caching is turned off, the Java SE consistently performs better with MySQL than with PostgreSQL. In contrast, if PostgreSQL is used with Perl information provider accessors, this performance is similar to the Java SE with MySQL. Therefore, it can be deduced that using Perl scripts where possible can dramatically reduce the average response time by 65%.

On the bottom graph of Figure 4.9, the throughput achieved by the MDS is shown. The

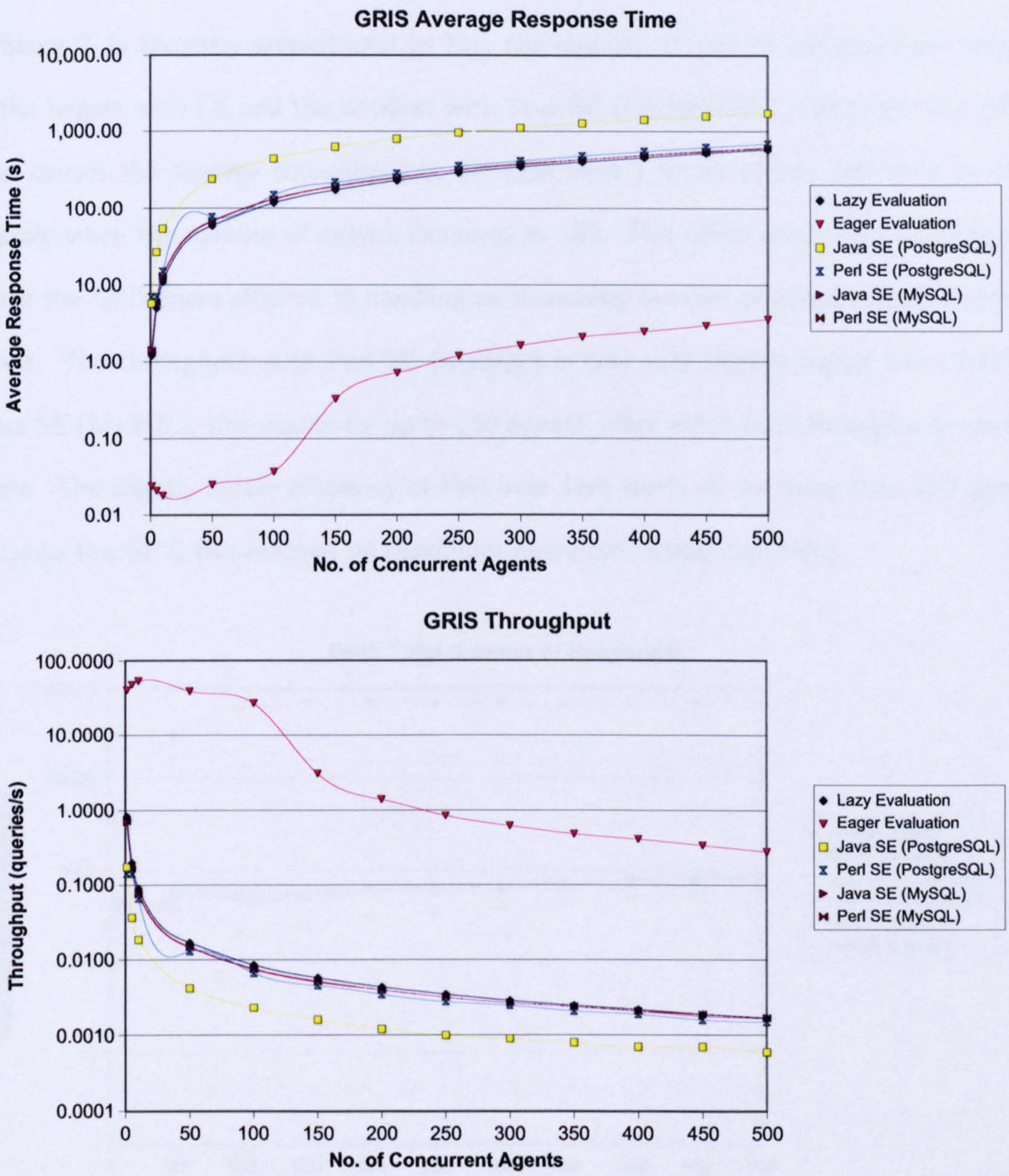

Figure 4.9: Experiment average response time and throughput.

relationship between throughput and response time is:

$$\mathcal{R}_T \propto 1/\mathcal{T} \tag{4.1}$$

Since $\mathcal{T}$ is inversely proportional to $\mathcal{R}_T$, the number of queries processed per second is the largest with EE and the smallest with Java SE (PostgreSQL). Caching in the GRIS also causes the average throughput to increase from 1 to 50 agents and then to drop sharply when the number of agents increases to 150. This effect shows that caching can make the GRIS more efficient in handling an increasing number of queries up to a certain point. The throughput with Perl SE (MySQL) is only very slightly higher than that for Java SE (MySQL); this occurs for up to 150 agents, after which both throughputs are the same. The slightly better efficiency of Perl over Java levels off for more than 150 agents because the GRIS has reached its maximum query processing capability.

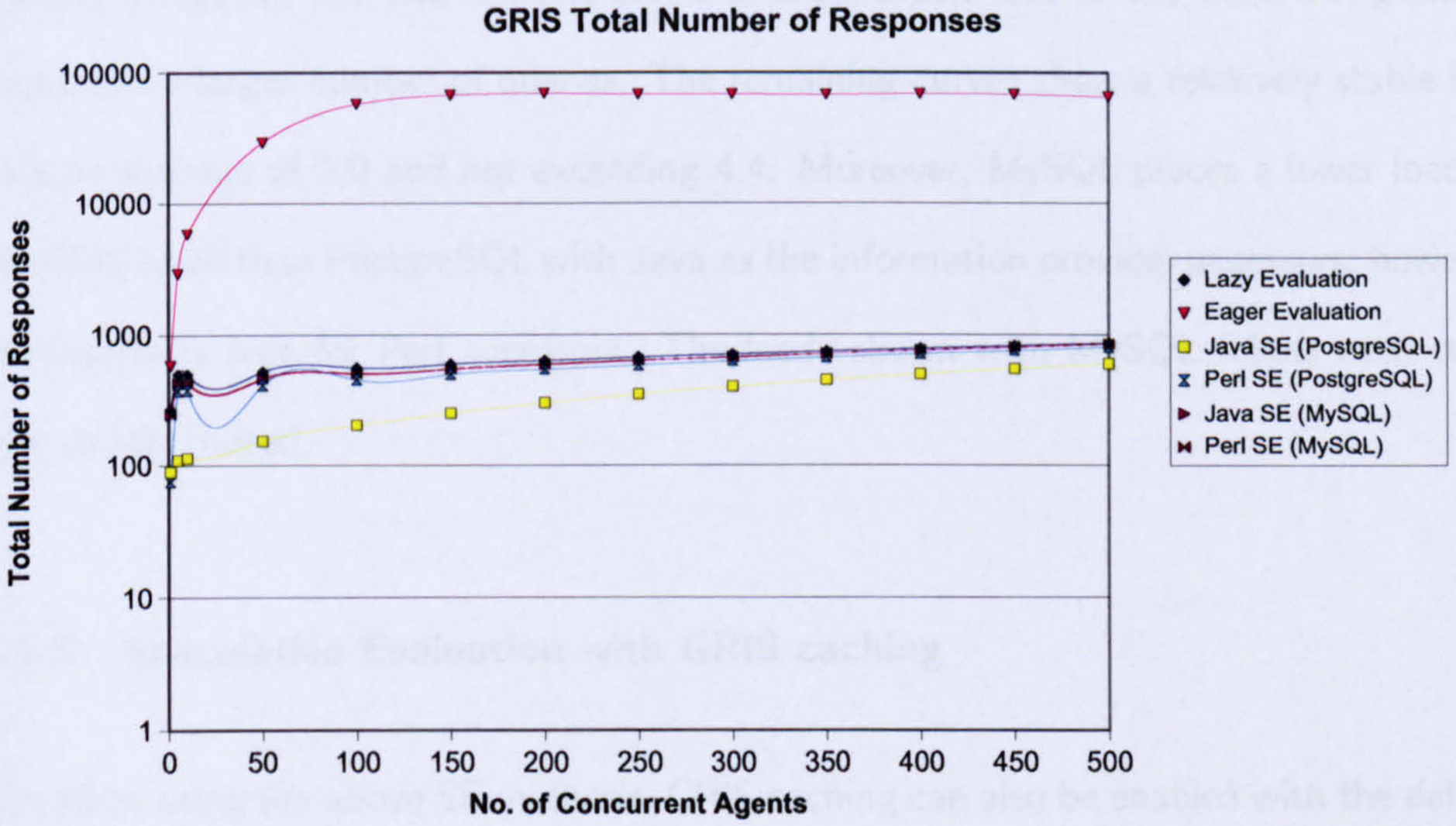

Figure 4.10: Experiment total number of responses.

The graph in Figure 4.10 shows the total number of successful query responses which return as the number of agents simultaneously querying the GRIS, increases. The three

curves involving EE display a gentle increase in the number of responses (about 5000) when up to 10 agents concurrently query the GRIS. Then, there is a sharp, steady increase in $\mathcal{R}_e$ when the number of agents increases up to 150. As this number continues to increase, $\mathcal{R}_e$ stabilises at around 65000. Again, caching in the GRIS enables more queries to be serviced; however, for more than 150 agents, the GRIS has reached its saturation point and it cannot process more than about 66000 queries.

The rest of the curves, apart from Java SE with PostgreSQL, show comparable behaviour with $\mathcal{R}_e$ ranging from 76 to 1000. Because the average response time for Java SE (PostgreSQL) was significant, its $\mathcal{R}_e$ struggles to increase beyond 600 for 500 agents.

Figure 4.11 shows the logarithmic scale graphs for the load averages on the GRIS node for 1 minute and 5 minutes. Querying the GRIS using EE places an increasing load on the GRIS with the number of agents increasing up to 150. This behaviour can be explained by the sharp increase in $\mathcal{R}_e$ seen in the previous graph. But for any further increase in the number of agents, the load remains stable at around 10.0 due to the GRIS being unable to process a larger number of queries. The remaining curves show a relatively stable load with an average of 2.0 and not exceeding 4.4. Moreover, MySQL places a lower load on the GRIS node than PostgreSQL with Java as the information provider accessors; however, the inverse is true for Perl accessors. The loads shown with MySQL follow each other very closely indeed.

4.5.5 Speculative Evaluation with GRIS caching

As well as using the above SE methods, GRIS caching can also be enabled with the default MDS values. The caching effect is illustrated in the bar chart below.

Figure 4.12 shows the averages for each performance metric; while GRIS caching reduces the average response time, it increases the average throughput, total number of responses and load averages.

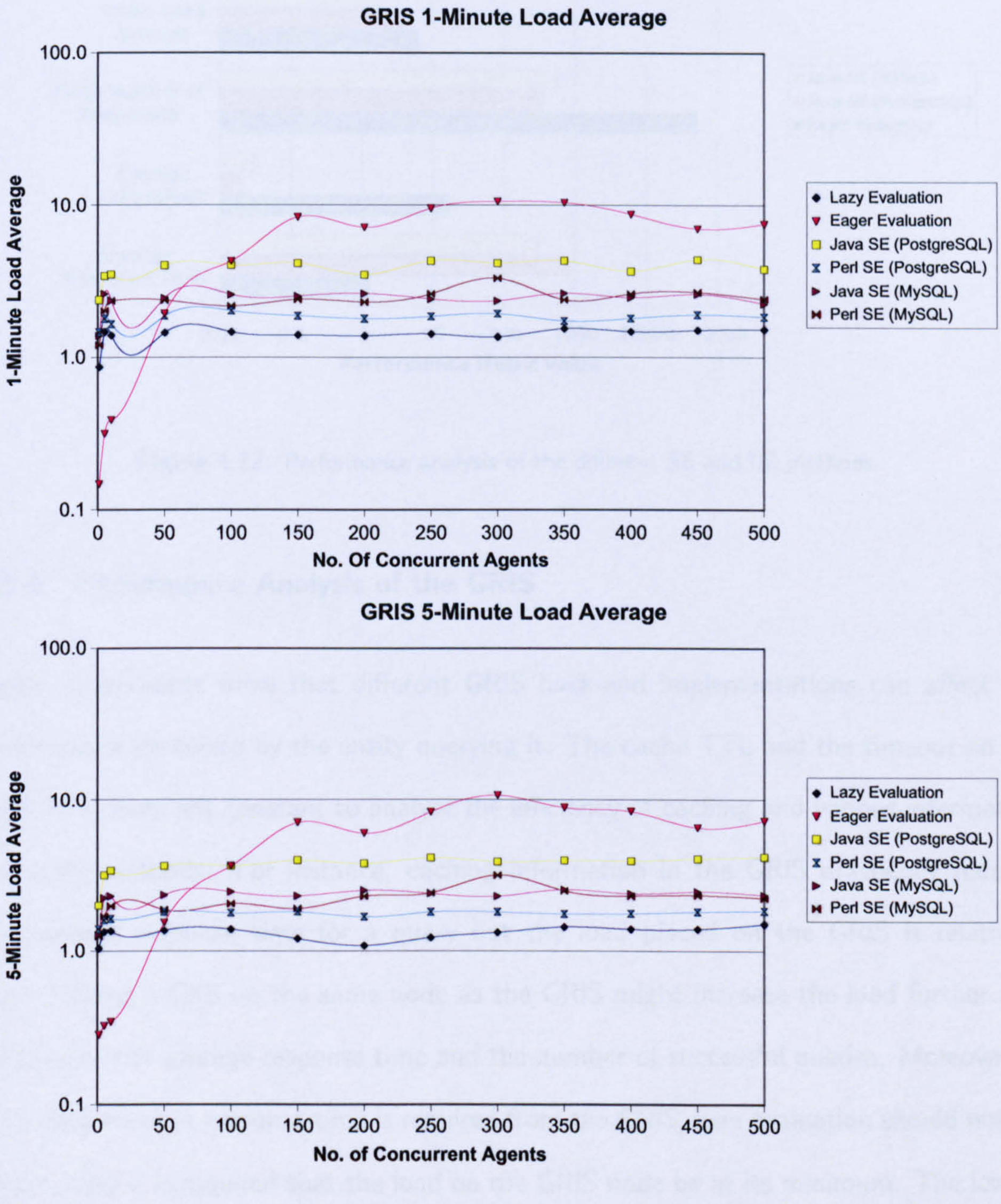

Figure 4.11: Experiment load averages.

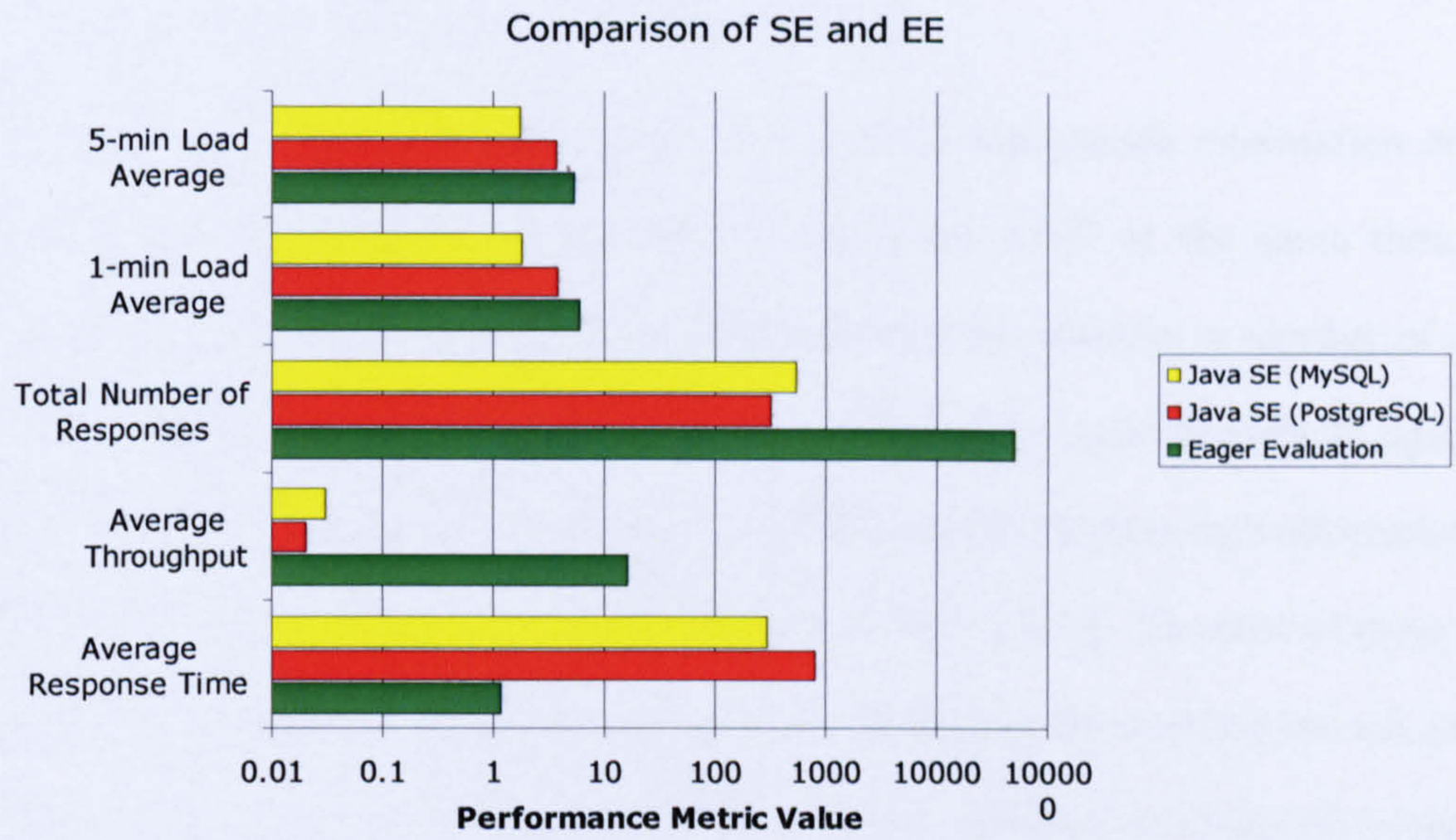

Figure 4.12: Performance analysis of the different SE and EE methods.

4.5.6 Performance Analysis of the GRIS

These experiments show that different GRIS back-end implementations can affect the performance perceived by the entity querying it. The cache TTL and the timeout on the GRIS have been left constant to analyse the efficiency of caching and various information gathering methods. For instance, caching information in the GRIS drastically reduces the average response time for a query but the load placed on the GRIS is relatively high. Having a GIIS on the same node as the GRIS might increase the load further and deteriorate the average response time and the number of successful queries. Moreover, if a minimal average response time is required from the GRIS, lazy evaluation should not be used, unless it is required that the load on the GRIS node be at its minimum. The lowest load during the experiment was with LE and it stayed relatively constant at approximately 1.5.

The advantage of SE is to facilitate the information updating mechanism which happens periodically as opposed to when there is a cache miss. SE also ensures that data which is

still within its TTL, is readily available in a database. Nevertheless, SE with both MySQL and PostgreSQL are slightly less performant than LE.

Choosing the method by which an information provider will provide information depends on the forecasted number of agents that will query the GRIS at the same time. The experiments in this section showed the worst case scenario whereby a number of agents kept querying the GRIS over a period of 10 min. If less than approximately 45 agents are querying the GRIS at any one time, it would be preferable to use the methods involving EE because their loads and $\mathcal{R}_T$ on the GRIS node are at their lowest. The cost of these methods, however, is data which might be out-of-date. Therefore, the method chosen depends on both the simultaneous number of queries and the permissible margin of information staleness. For example, information which rarely changes like the operating system, has a greater margin than dynamic information. For more than 45 concurrent agents, the load with the EE methods increases dramatically and stays stable at a level much higher than the other methods, even SE. Again, there is a trade-off amongst $\mathcal{R}_T$, GRIS load and information freshness. If the agent requests the most up-to-date information, then LE should be used, but at a higher cost of larger $\mathcal{R}_T$, $\mathcal{L}_1$ and $\mathcal{L}_5$, than EE.

The choice of implementation language for the information providers and the database in the SE methods also affect performance. Perl generally provides a smaller $\mathcal{R}_T$ than Java, and MySQL is more performant that PostgreSQL.

4.6 Performance Prediction of MDS2 Queries

Further experiments have been carried out, which are similar to those of the GRIS, but where clients query the GIIS instead. Based on the GIIS performance data collected, the values for the performance metrics are predicted using different algorithms in this section. Thus, past performance observations can be used to characterise the future performance of the GIIS qualitatively, allowing Grid middleware built on these services to be more

predictable. Several different predictors are discussed and the way they are applied to previous performance data is analysed.

4.6.1 Predictive Methods

It is difficult to observe a trend in the behaviour of the GIIS; therefore, a number of predictive methods are applied to past GIIS data. In addition, the characterisation of the performance of the MDS does not take into account the network load or the Grid topology. The approach taken is to formulate a prediction for the GIIS performance, for the benefit of a Grid application. A performance methodology is therefore developed to choose the most appropriate GRIS evaluation method and for predicting the cost of queries. The performance prediction of a query from observations of past queries can be used by the end user, that is the agent, which is interested in the resource discovery end-to-end performance. This performance information can be used in its other functionalities, including metascheduling and contributing to the guarantee of quality-of-service contracts.

The performance prediction of a query to the MDS is based on collecting performance information for each query that is made to the GIIS and then applying predictive methods to the previous observations. The gathering of performance information does not affect the behaviour of the MDS in any way. The different predictors [25] used to estimate future query times are:

- **Last observation** The most recent, single performance observation value is taken as the prediction. The last performance value is most likely to reflect the behaviour of future queries.

$$P_n = V \tag{4.2}$$

- **Sample average** The prediction is the mean average of the past performance values within a sample set. This set is defined by a sliding window of size x, which corresponds to the x most recent observations. Not all the performance values are

used for the average as old values become less relevant. This predictor is used when performance information is produced on a regular basis. However, given a fixed performance data set, an average can be used with a maximum window size.

$$P_n = \frac{\sum_{i=1}^{x} V_i}{x} \tag{4.3}$$

- **Low pass filter** Recent performance data constitutes a better predictor than older data. Subsequently, this predictor uses an exponentially degrading function to obtain an average of the recent performance behaviour of the MDS. This is achieved by using the *low pass filter* formula:

$$P_n = (w \bullet P_{n-1}) + ((1 - w) \bullet V) \tag{4.4}$$

where

P_n is the prediction and the new value of the low pass filter

P_{n-1} is the previous filter value

V is the most recent performance observation value

w is the weighting parameter and is a value between 0 and 1

The value of *w* is 0.95 as suggested in [25], thus decreasing the value of the weight as observation values grow older and increasing the prediction accuracy.

- **ARIMA models** Due to the fact that the observed data is *stationary* i.e it varies about a fixed level, *ARIMA* (autoregressive integrated moving average) [7, 8] models are used to project the data to produce forecasts. The two most adequate models have been identified and are the AR(1) and AR(2) models. In the AR(1) model, forecasts for the next value depend on the observations in the previous time period; whilst in AR(2) models, forecasts of the next value depend on observations in the two previous time periods.

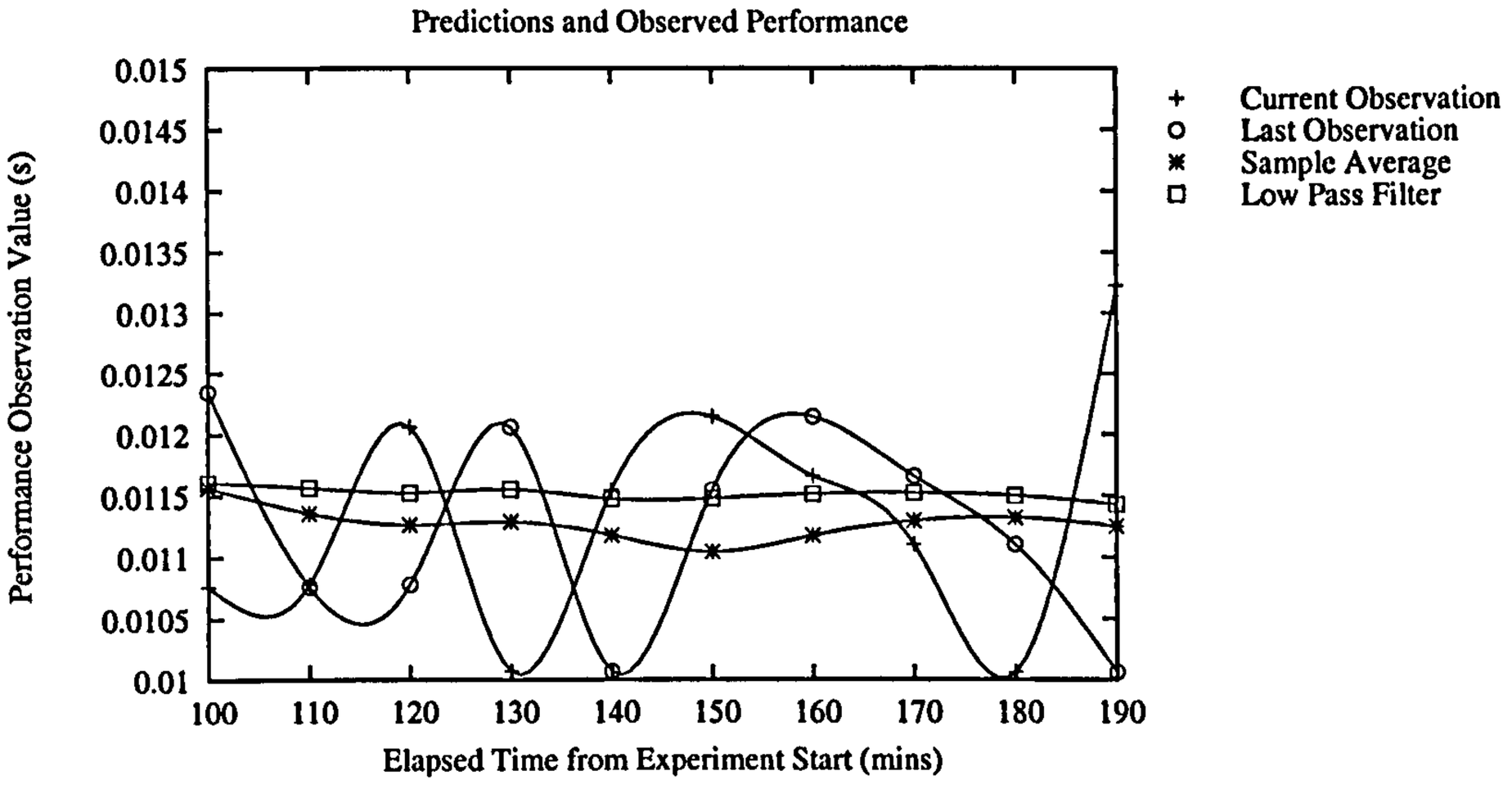

Figure 4.13: Actual observed data and predictions.

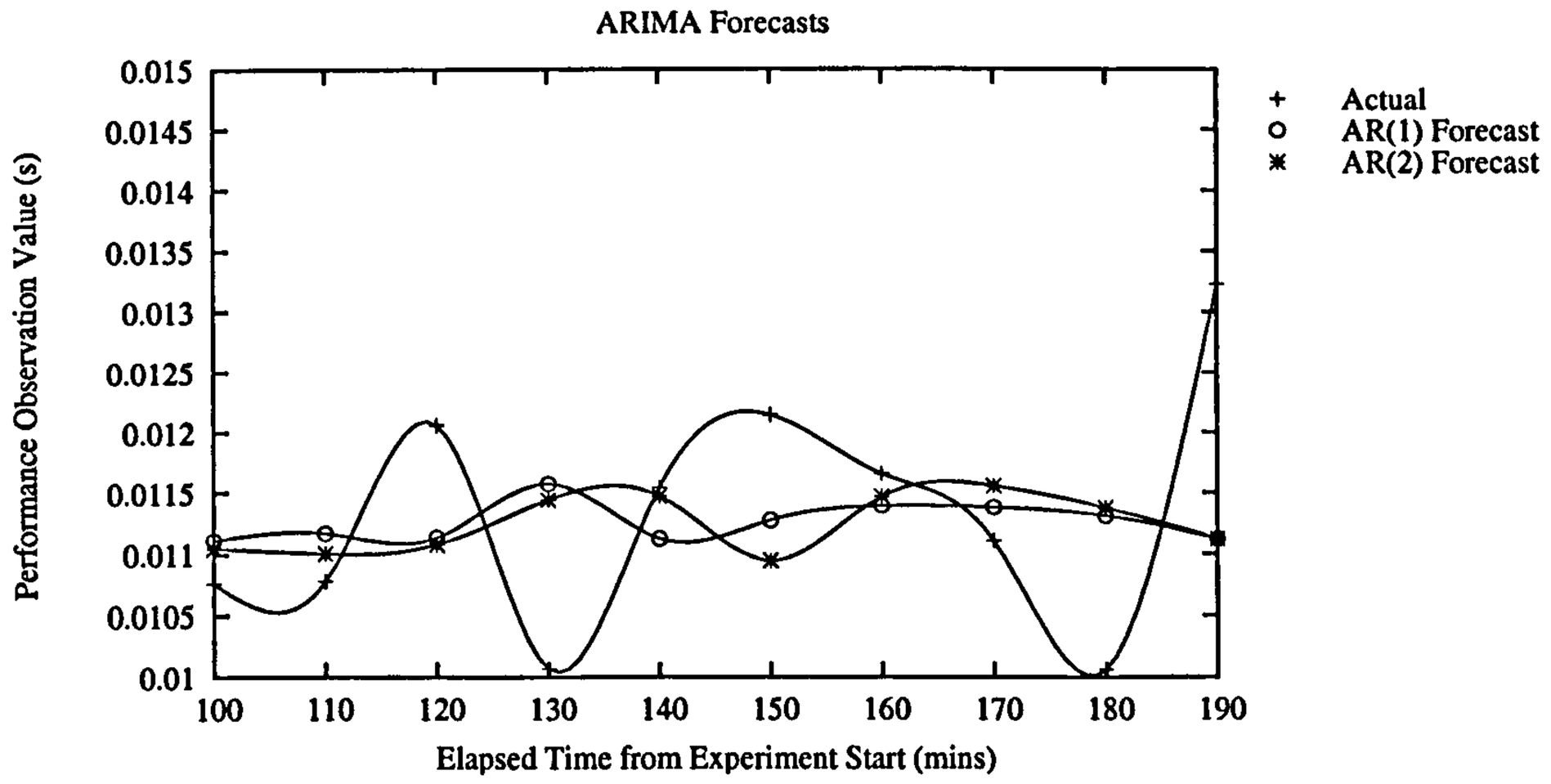

Figure 4.14: Actual observed data and ARIMA forecasts.

4.6.2 Query Performance Prediction Results

Figures 4.13 and 4.14 show the results obtained when the average time for a query is predicted using the different predictive methods. In addition, the actual performance observation data is also shown. In the experiment, one agent repeatedly queries the GIIS using the EE implementation, and the average response times collected. A sliding window of ten performance observation values are used for the *sample average* predictor. Moreover, the mean of the sample data set is used as the initial prediction for the *low pass filter* predictive method. Query performance prediction is started when 100 min have elapsed since the beginning of the experiment.

The one-step ahead AR(1) and AR(2) forecasts which are made over an increasing time series, are shown in Figure 4.14. These two models have been checked for residuals and are considered adequate for forecasting. The graphs show that the two kinds of predictions closely fit the actual observed time series.

The predicted and actual values for $\mathcal{R}_T$ at each recorded time, which are shown in Figures 4.13 and 4.14, indicate various levels of prediction accuracy. For instance, after 120 min have passed since the start of the experiment, the observed data is 0.012s, *last observation* predicts 0.010s, and *sample average*, *low pass filter* and both *ARIMA* methods predict a time of 0.011s. Furthermore, since the observed data tends to vary greatly from one value to the next, the *last observation* prediction which follows it, is rather inaccurate. On the other hand, the *sample average* and *low pass filter* predict values which fluctuate rarely.

	RMSE ± 95% Confidence Interval
Last Observation	0.001492 ± 0.00055
Sample Average	0.005585 ± 0.00207
Low pass filter	0.002970 ± 0.00104
AR(1) Forecast	0.001022 ± 0.00043
AR(2) Forecast	0.001039 ± 0.00043

Figure 4.15: 95% confidence interval values for the different predictors.

$$RMSE = \sqrt{\frac{1}{N}\sum_{x}(O_x - C_x)^2} \tag{4.5}$$

where

N is the data sample set size

O is an observation value

C is a calculated value via prediction

Figure 4.15 shows the 95% confidence values and RMSE (root mean square error) as shown in Equation 4.5, for the different predictive algorithms. The 95% confidence interval values shown, specify the range of values (between the lower and upper limits) within which the difference of the means of the actual observed data series and the data series for the predictor, may lie. From the confidence interval values shown, the difference in the performance of the various predictors is statistically insignificant, with AR(1) and AR(2) having the lowest RMSE and confidence interval values.

4.6.3 Summary

The approach taken in this section is to predict the behaviour of the MDS from a Grid application's point of view, based on past performance data. These predictions are used to help the application decide which GIIS to choose for sending queries. This informed choice further contributes to guarantee quality-of-service in the use of Grid middleware. To do so, the performance achievable when queries are sent to a GIIS, has been analysed and compared with that of a GRIS. Several scenarios have been set up with different information providers and GRIS back-end implementations. The experiment results demonstrate that caching i.e EE, is required at the higher levels of the MDS hierarchy for an acceptable level of performance to be obtained. Furthermore, a better performance is obtained when a GIIS is queried, rather than a GRIS when caching is enabled in the GIIS and is at least 60 s. The value of the cache TTL depends on the expected number of users concurrently querying the GIIS.

Using past MDS performance observation data, several predictive algorithms are implemented and the experiment results analysed. It has been found that even though the AR(1) and AR(2) predictive methods displayed the smallest difference from the actual data set, the other methods are still comparatively accurate.

4.7 Queries of Varying Complexity

Queries which client applications ask can be of various complexity. For example, searches can return all the information within a scope, or incorporate a number of criteria that must be satisfied. In addition to the resource discovery requirements for Grid applications, there is also the need to provide reliable performance and the delivery of quality-of-service. This performance requirement calls for a series of benchmarks to be developed to assess and characterise the level of response a Grid Information Service (GIS) would present to searches. In order to achieve this goal, this section of the thesis focuses on the development

and application of a set of benchmarks that measures qualitatively and quantitatively the overhead of using the Globus Toolkit's Monitoring and Discovery Service (MDS2). Using various Grid Resource Information Service (GRIS) data provenance mechanisms, these benchmarks are run and performance data is collected when a Grid Index Information Service (GIIS) receives queries of greatly varying complexity. Several performance metrics are defined and the performance results are evaluated in terms of the quality-of-service (QoS) achieved; these results provide an insight into the performance, scalability, reliability and robustness of the GIS studied. The findings can therefore help make recommendations for future MDS setup and configuration, as well as Grid application tuning.

Client applications making use of a Grid Information Service like the MDS, would typically assess the efficiency with which the system can be used. An important issue is the expected response time for a query. The answer depends on many factors, including the physical features of the machine on which the MDS is running, the load on the machine, and the characteristics of the resources on which information providers are running. Another decisive factor is the level of complexity of the query posed. This section analyses the behaviour of the GIIS in responding to a range of queries via several typical scenarios, and will run benchmarks from the point where the query reaches the MDS, to when the client receives the response. Consequently, whilst network monitoring is indeed part of this resource discovery process, this section concentrates on the characterisation of the behaviour of the MDS alone, without the external influence of network conditions. This research work therefore pinpoints the performance of the actual MDS query processing solely, which is best done and examined using a LAN.

4.7.1 Query Benchmarks

Previous work on the performance evaluation of the MDS, described in the previous sections, consisted in a Grid application requesting all data objects in a particular domain. Nevertheless, clients can issue more refined searches using the LDAP query language.

Here, the performance of refined, disparate searches are investigated, and compared with more general queries.

The LDAP query language which is used with the MDS, is fairly expressive, allowing a wide range of queries to be issued against it. The proposed benchmark includes a variety of queries: simple versus complex searches, those which return a small data set versus a larger one and searches which apply filtering to the result set. The aim is to select query types that are representative of the resource discovery behaviour of Grid applications. The data which is the subject of the queries in the benchmarks, is dynamic and it therefore changes by the order of seconds. This dynamic data renders the benchmarks more realistic as Grid applications are more likely to query the MDS repeatedly for changeable data, rather than static data. Subsequently, it is important to study the performance of the information service when such client workload is imposed. The following sub-section details the query benchmarks used.

Query Combinations

The benchmarks include eight queries which return various attribute combinations about the local scheduler Titan, which is interfaced by GRAM, as well as its job information. More details about the configuration of the local scheduler, and how data is procured from it, were given in a previous section. The queries are differentiated according to these features: scoping, objectclass selection, number of criteria and the type of boolean operator.

- **Scoping** This term defines the particular sub-tree to which the search is applied. Specifically, it allows the GRIS or the GIIS to be queried. Similar research has been carried out before where queries had different scopes [68], but where the query remained constant.

- **Objectclass** This special attribute defines groups of entries with certain required

and allowed attributes. It is usual to restrict the searches to objectclasses of interest in order to minimise the query response time. When objectclasses are used in the experiments, they pertain to data produced by the local scheduler.

- **Number of criteria** Searches can include any number of conditions that must be satisfied for any entry to be returned as the result. The experiments aim to examine the difference in performance of searching for information with numerous filters.

- **Boolean operations** The above criteria can be composed in diverse combinations using Boolean operators; these are the AND, the NOT and the OR operators. Using these operations, search filters are specified in polish notation. This reflects the complexity of searches against the MDS.

An example query to the information service about a batch scheduler is "find all the hosts in a virtual organisation [37] for which the scheduler's last job finishing time is greater than or equal to x". Such a search will require a filter and a condition applied when accessing the MDS. Using LDAP, this query would be as follows:

```
grid-info-search -b "mds-vo-name=VOname, o=Grid"
"(&(objectclass=MdsBatchScheduler)(Mds-Scheduler-FinishTime>=x))" dn
```

A summary of the queries is given in Table 4.1:

Table 4.1: Summary of query types.

return all entries
return one objectclass' entries
filter based on one objectclass and one condition
filter based on one objectclass and 2 conditions linked with OR operator
filter based on one objectclass and 3 conditions linked with OR operator
filter based on one objectclass and 2 conditions linked with AND operator
filter based on one objectclass and 3 conditions linked with AND operator
filter based on one objectclass and one NOT condition

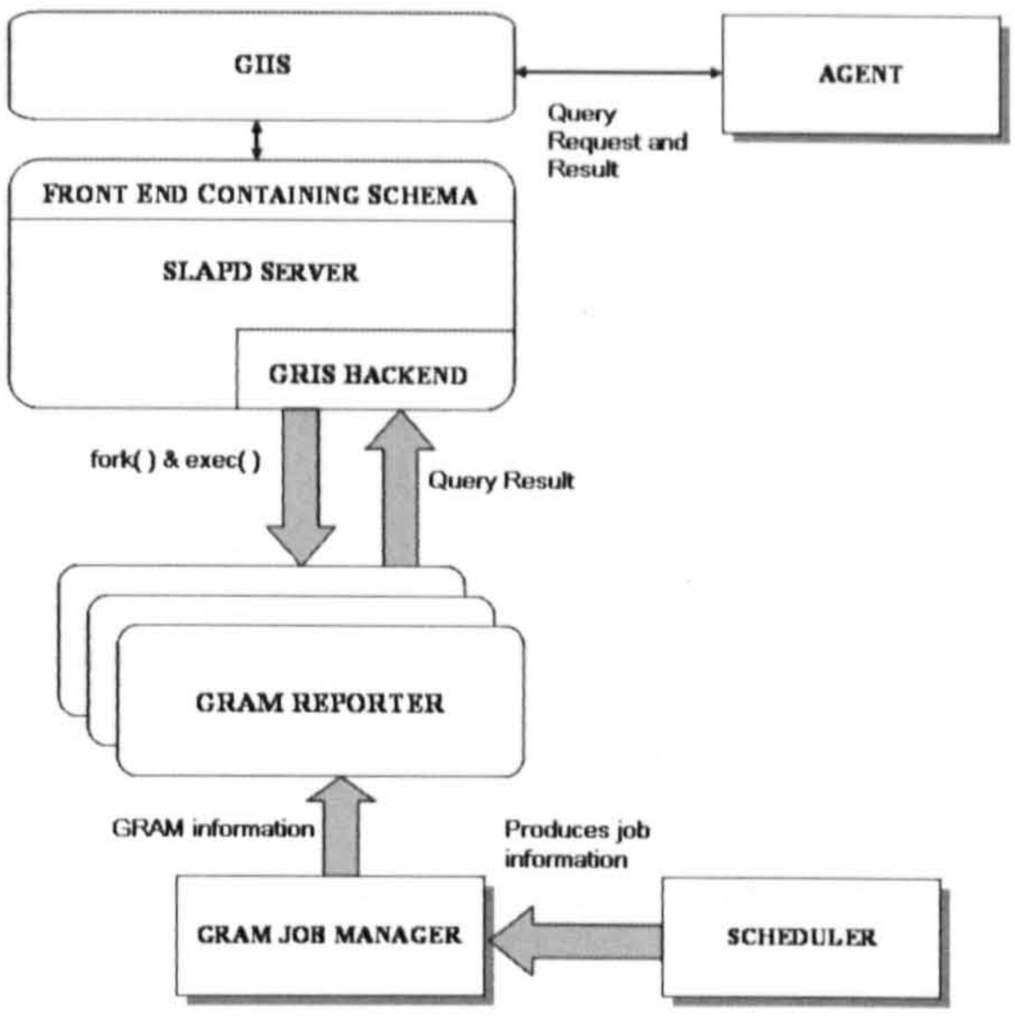

Figure 4.16: The GRAM reporter publishes scheduler job information to the MDS.

The interaction of the GRAM reporter with other MDS components is shown in Figure 4.16. The Grid Information Service is configured so as to allow the GRAM reporter to publish information from the Titan job manager. The nature of such information is different from other hardware information in that it is highly dynamic. The rate of change of data is typically of the order of seconds. The experiments which have been carried out consist in searching the MDS for GRAM information.

4.7.2 Experimental Setup

The experiments were carried out on a Grid testbed at the University of Warwick and were based on MDS 2.4 which is part of the Globus Toolkit 3 and is the latest MDS 2.x version which includes fixes for improved scalability, reliability, and stability. The core information providers have been extended with GRAM reporters which publish dynamic local scheduling data into the MDS. Thus, five GRAM reporters have been configured to produce data including the job deadline and the number of jobs in the local queue. The

local scheduler Titan itself, was running on another host on the same LAN. That host had the following specifications: a Linux kernel 2.4.18-14 machine with a 1.8 GHz processor and 512 MB RAM. Across the various experiments, the following Grid application setup was maintained where agents were written using the Java CoG Kit libraries.

Agents representing Grid client applications, make request queries to the MDS; these queries are sent from a set of ten machines (Γ_1 .. Γ_{10}). With a maximum of 300 agents simultaneously making queries over a period of ten minutes, the desired effect was to load-balance the queries and to sustain the MDS querying. The maximum number of agents attributed to one machine is therefore 30 (same set-up as in the first footnote in Section 4.5.3). In these experiments, the client machines are connected to the GIIS host. The time taken for each request to be serviced is measured and an average response time is calculated. Moreover, every agent sleeps for one second before sending the next request. To test the scalability of the MDS, a GIIS and a GRIS were both set up on a Linux kernel 2.4.18-14 machine (M_1) which has a 1.9 GHz processor and 512 MB RAM. In these sets of experiments, the performance of the MDS will be monitored as it receives an increasing number of similar requests. Performance data was collected from a set of ten experiments, and the average results are shown. This performance study will therefore allow a more realistic prediction of the behaviour of the MDS where agents search for specific data.

4.7.3 GRIS Backend Implementations

Using the evaluation methods explained above, a number of implementations can be configured and set up for the GRIS backend. Experiment results from previous sections in this chapter, indicate that the choice of implementation language for the information providers and the database used in the speculative evaluation methods greatly influence performance seen by the application, as well as the overhead on the MDS server. In

general, Perl and MySQL are more performant than Java and PostgreSQL respectively. Consequently, in this section, the GRIS backend implementations which will most likely meet QoS user requirements, are compared when various queries are issued to the MDS. The backend implementations used are therefore:

- **Lazy evaluation (LE)**

  The GRIS cache TTL is equal to zero. On the receipt of each query, the GRIS launches its core information providers.

- **Eager evaluation (EE)**

  The GRIS cache TTL is not equal to zero (default values) and the information providers are invoked when the cache is expired. The cache is also filled with the new data.

- **Perl speculative evaluation (MySQL)**

  The GRIS cache TTL is equal to zero and the information providers write their data to a MySQL database on average every minute. Moreover, the information provider accessors are written in Perl.

Performance Metrics

Similar performance metrics as in previous sections, are used for consistency and comparison. The full list of performance metrics are thus:

- Average response time in seconds ($\mathcal{R}_T$). This is the average time from sending out a query and receiving the response, across all the successful queries. The theoretical maximum is the length of the experiment, which is 10 min.

- Average throughput in terms of the number of queries answered per second ($\mathcal{T}$). This is a server-side metric which demonstrates whether the MDS scales with an increasing number of concurrent queries.

- Total number of successful query responses ($\mathcal{R}_e$). A successful query is defined that occurring when MDS results return to the particular client, without timing out. This metric is a server-side one, showing the total number of queries serviced throughout the whole experiment duration.

- 1 min load average ($\mathcal{L}_1$). The load average is indicative of the MDS being under heavy usage, which increases the average response time and time-outs.

- 15 min load average ($\mathcal{L}_{15}$).

- Percentage of free memory ($\mathcal{M}$). The required memory increases with the number of queries, and is indicative of the usage of the MDS.

Experimental Disparate Queries

For the experiments, the application of query scoping entails that the data from a single virtual organisation VO is searched. Thus, the GIIS is queried for dynamic job information. Table 4.2 below shows the details of the disparate queries used. The experiment results in the following section show how these queries with different filter conditions affect the client- and server-side performance of the MDS.

4.7.4 Query Type Experiment Results and Evaluation

The speculative evaluation experiments were such that the GRAM reporter cache times were 30 s for each provider. This meant that job information was being published to the MySQL database every 30 s. The experiment results are now analysed to demonstrate any effect on the QoS experienced by clients querying the MDS and on the MDS host server itself. Additionally, the version of MySQL used was 4.0.1 and database caching was disabled.

Table 4.2: Queries used in the experiments.

Filter name	LDAP query
A	"(objectclass=*)"
B	"(objectclass=MdsSchedule)"
C	"(&(objectclass=MdsSchedule)(Mds-Scheduler-Deadline<=17132.0))"
D	"(&(objectclass=MdsSchedule)(|(Mds-Scheduler-Phenotype>=100) (Mds-Scheduler-Deadline>=500)))"
E	"(&(objectclass=MdsSchedule)(|(Mds-Scheduler-Phenotype>=100) (Mds-Scheduler-Deadline>=500)(Mds-Scheduler-Dominant-Type=0.0)))"
F	"(&(objectclass=MdsHost)(&(Mds-Cpu-Free-1minX100<=100) (Mds-Memory-Ram-freeMB<=500)))"
G	"(&(objectclass=MdsSchedule)(&(Mds-Scheduler-Phenotype>=100) (Mds-Scheduler-Deadline>=500)(Mds-Scheduler-Dominant-Type=0.0)))"
H	"(&(objectclass=MdsSchedule)(!(Mds-Scheduler-Dominant-Type=10.0)))"

Experiment Results with Lazy Evaluation

Figure 4.17 shows the average response time experienced by each agent for the duration of an experiment. Requesting and returning all data objects clearly increases the average response time; the increase is generally linear. For example, when the number of concurrent agents making the same queries with filter A reach 300, the average response time is approximately 5 seconds. The response times experienced when the other filters are used, indicate that using a filter can greatly decrease the average response time compared to requesting all data objects. Interestingly, the results for filters B to H show that the average response time is hardly affected by the increasing number of queries. The response time stays around 0.025 seconds as the number of concurrent agents increases to around 200. When a greater number of simultaneous agents issue queries to the MDS, filters F and H show an upward trend in the average response. This indicates that the AND and the NOT boolean operators tend to impact on query performance for more than 200 agents. Additionally, from the group of better performing filters B to H, filter F has an overall slightly higher average response time.

The average throughput graph in Figure 4.18 has an inverse relation with the average

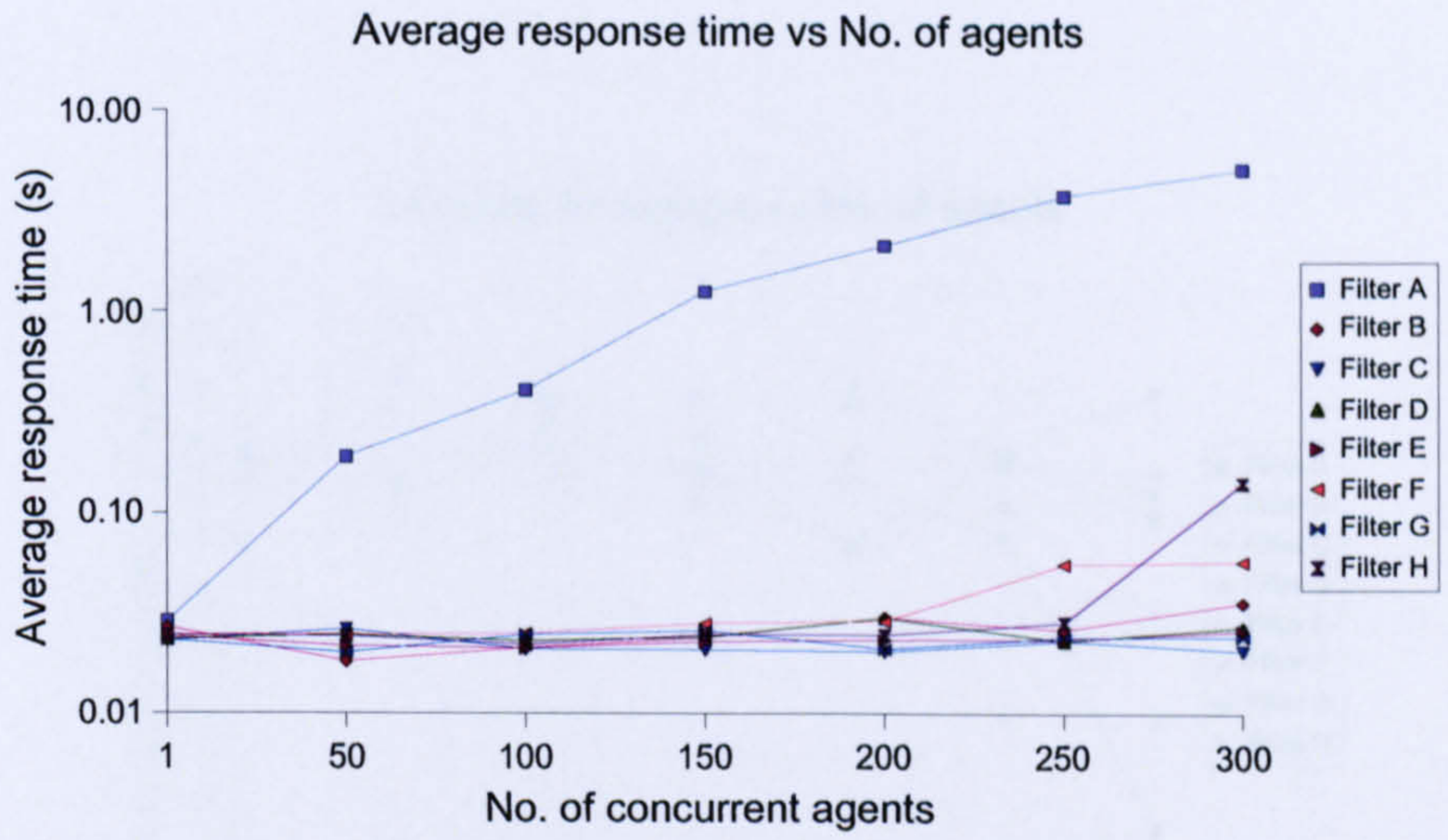

Figure 4.17: Comparison of average response time with disparate queries (LE).

response time graph. It is seen that with filter A, the MDS can process approximately 35 queries per second when only one agents repeatedly issues queries. But this throughput quickly decreases in a logarithmic fashion, only processing around 0.2 query per second with 300 simultaneous agents. Furthermore, filter C displays the consistently highest average throughput, while filter H displays the worst (at 300 agents). This implies that searching the MDS with one objectclass and one condition results in reliable performance. Filter E (OR operator with three conditions) also displays a consistent high throughput, not allowing less than about 40 queries to be processed per second.

Figure 4.19 shows the total number of successful queries that have been serviced in the 10 min period for an experiment. With filter A, the total number of responses increases with an increasing number of concurrent agents, but it cannot improve on about 16000 queries per experiment. The rest of the filters exhibit an almost linear increase in the total number of responses with more concurrent agents. However, filters F and H return slightly fewer responses for more than 250 simultaneous agents. This can be explained by the longer processing time for the AND boolean operator (with two operands) and the NOT operator.

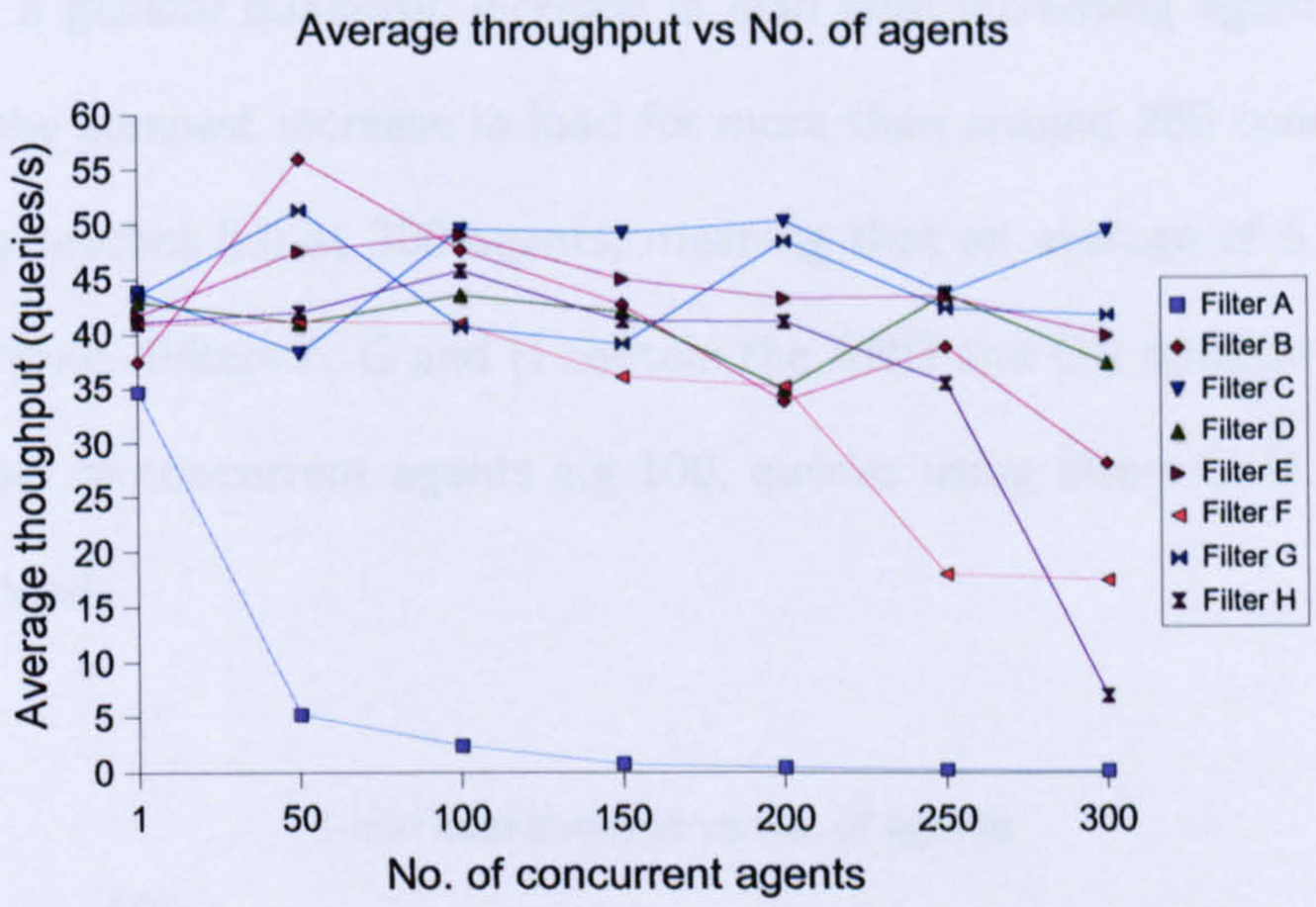

Figure 4.18: Comparison of average throughput with disparate queries (LE).

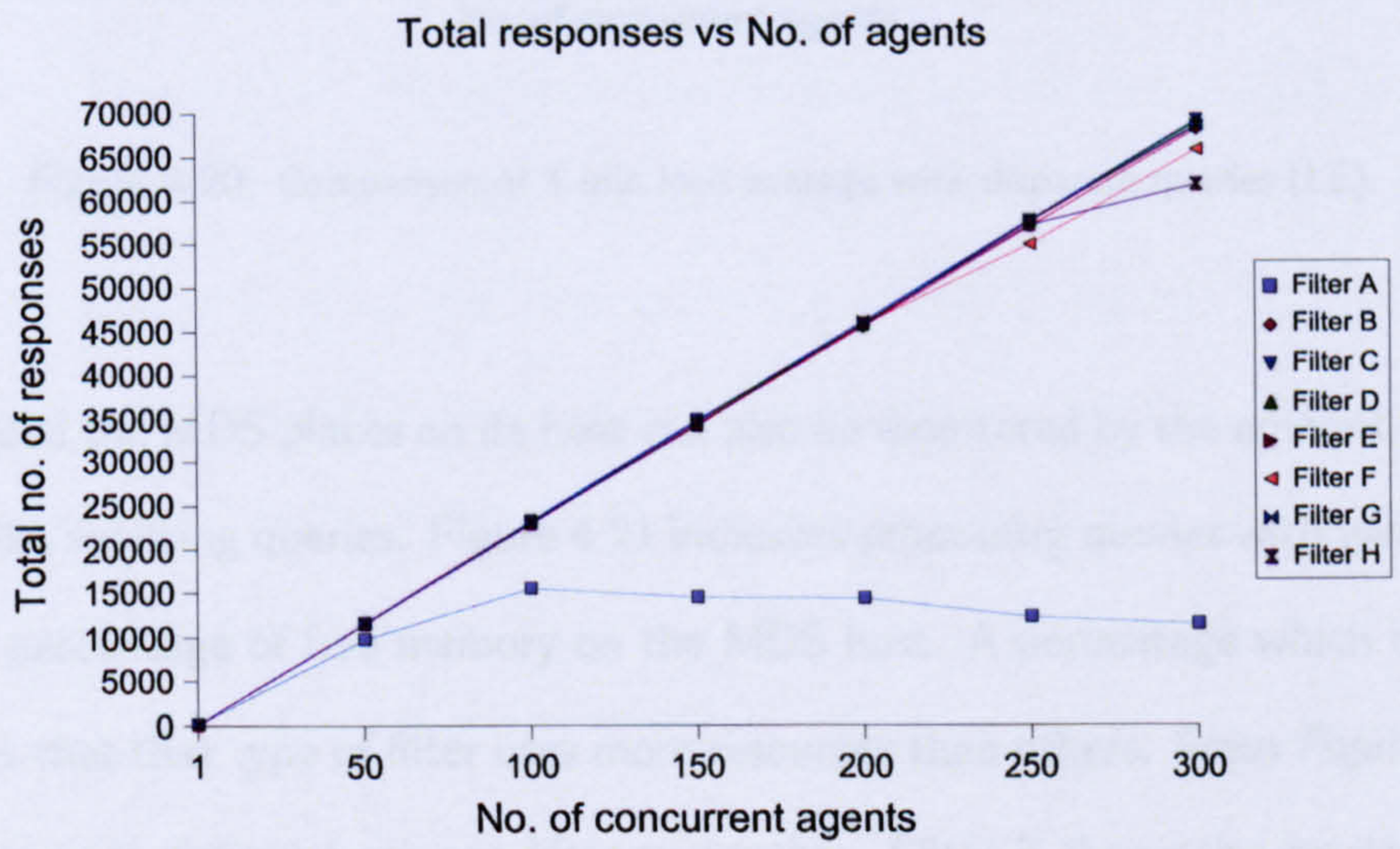

Figure 4.19: Comparison of total number of responses with disparate queries (LE).

The performance overhead of the GIIS host is shown in Figure 4.20. Since both the 1 min and 15 min load average graphs are identical, only the 1 min averages are shown. All the curves indicate a general quadratic increase in load with increasing agents. Filters F, G and H display the steepest increase in load for more than around 250 concurrent agents. The load nearly reaches 5.0 at 300 agents; meaning that an average of 5 jobs are in the run queue at a time. Filters F, G and H contain the AND and OR boolean operators. For a smaller number of concurrent agents e.g 100, queries using filters C, D and E result in the least CPU load.

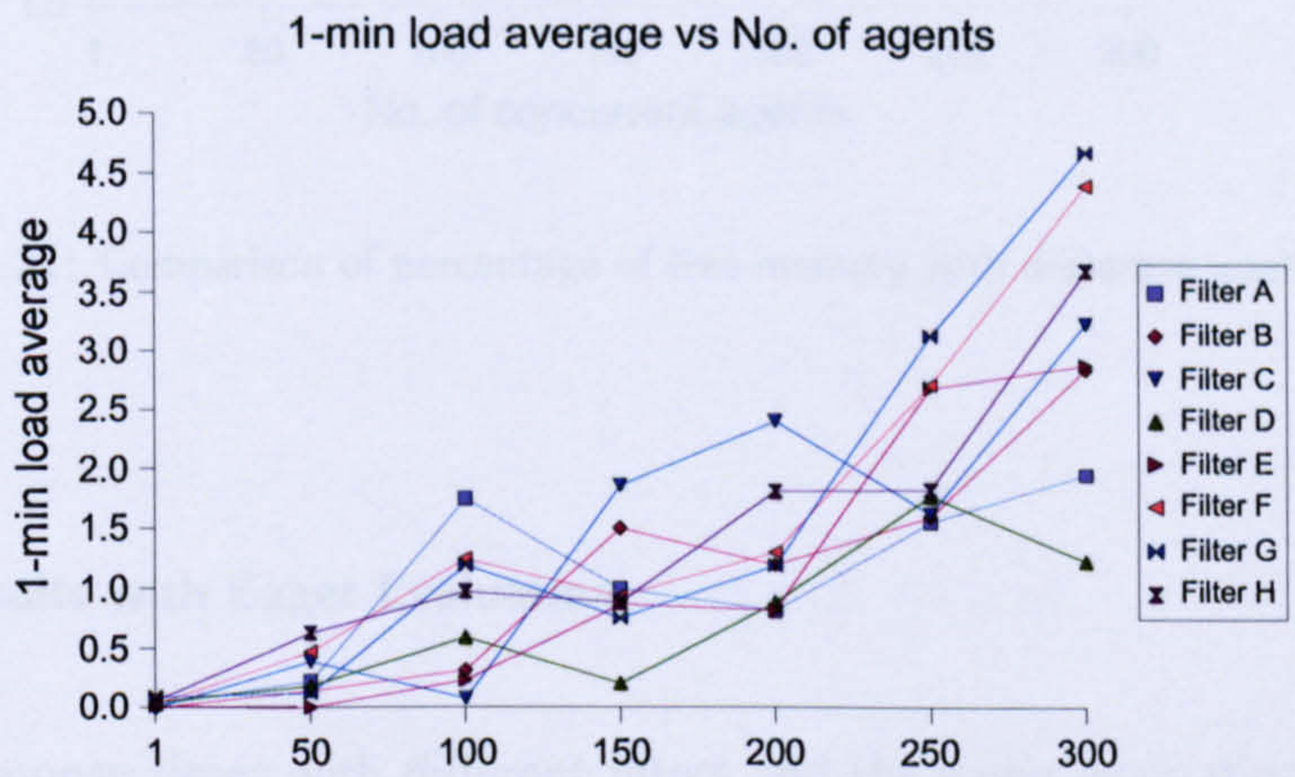

Figure 4.20: Comparison of 1 min load average with disparate queries (LE).

The overhead the MDS places on its host can also be monitored by the amount of memory it uses whilst servicing queries. Figure 4.21 indicates processing queries with various filters, affect the percentage of free memory on the MDS host. A percentage which is relatively low implies that that type of filter uses more resources than others. From Figure 4.21, the memory usage is distinct from one filter to another. Filter F shows the greatest increase in the percentage of free memory. Filters G and H both results in the same amount of memory to be used as the number of agents increase. This indicates that both the NOT and the AND operator filters utilise relatively less memory than others. These results

indicate that the number of data objects which is returned influences the amount of memory which is free: the more data objects, the smaller the percentage of free memory.

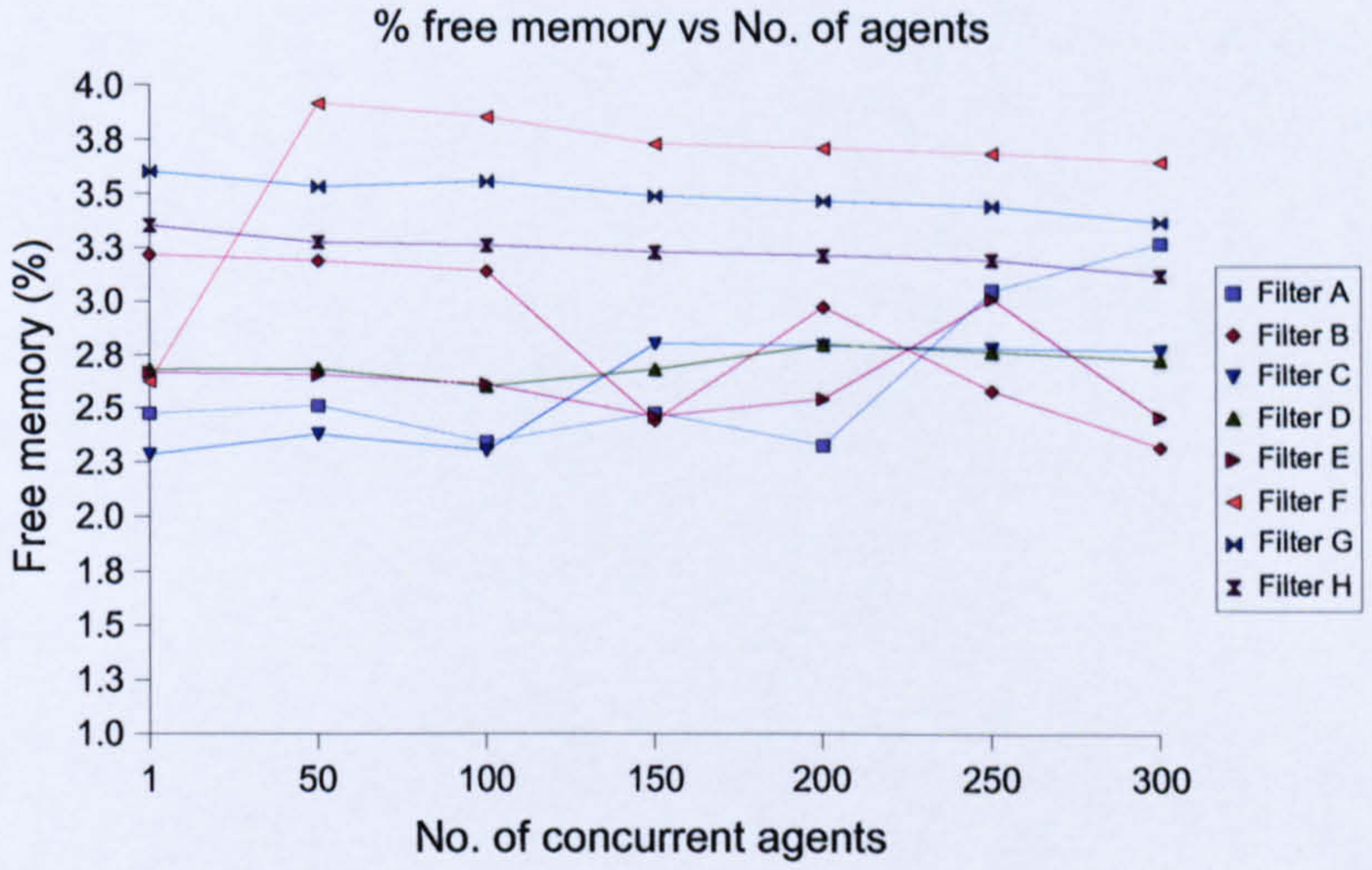

Figure 4.21: Comparison of percentage of free memory with disparate queries (LE).

Experiment Results with Eager Evaluation

The average response times with different filters and the eager evaluation GRIS method, are shown in Figure 4.22. There is a clear demarcation between results with filter A and the other ones. This indicates that specifying an exact search allows for quicker responses than no filter at all. The average response times for queries with filters B to H stay relatively fixed at 0.030 s, even with increasing agents. However, filter F responses increase more quickly than others for over 200 agents. This can be explained by the use of the AND boolean operator.

When the evaluation method is the eager one, the throughput graph in Figure 4.23 shows that returning all the data objects (filter A) significantly decreases the number of queries processed per second. The filters which show the best highest throughput are the one condition filter, one objectclass and the OR operator. Filter F indicates the least throughput average, allowing this conclusion to be reached: the AND operator reduces

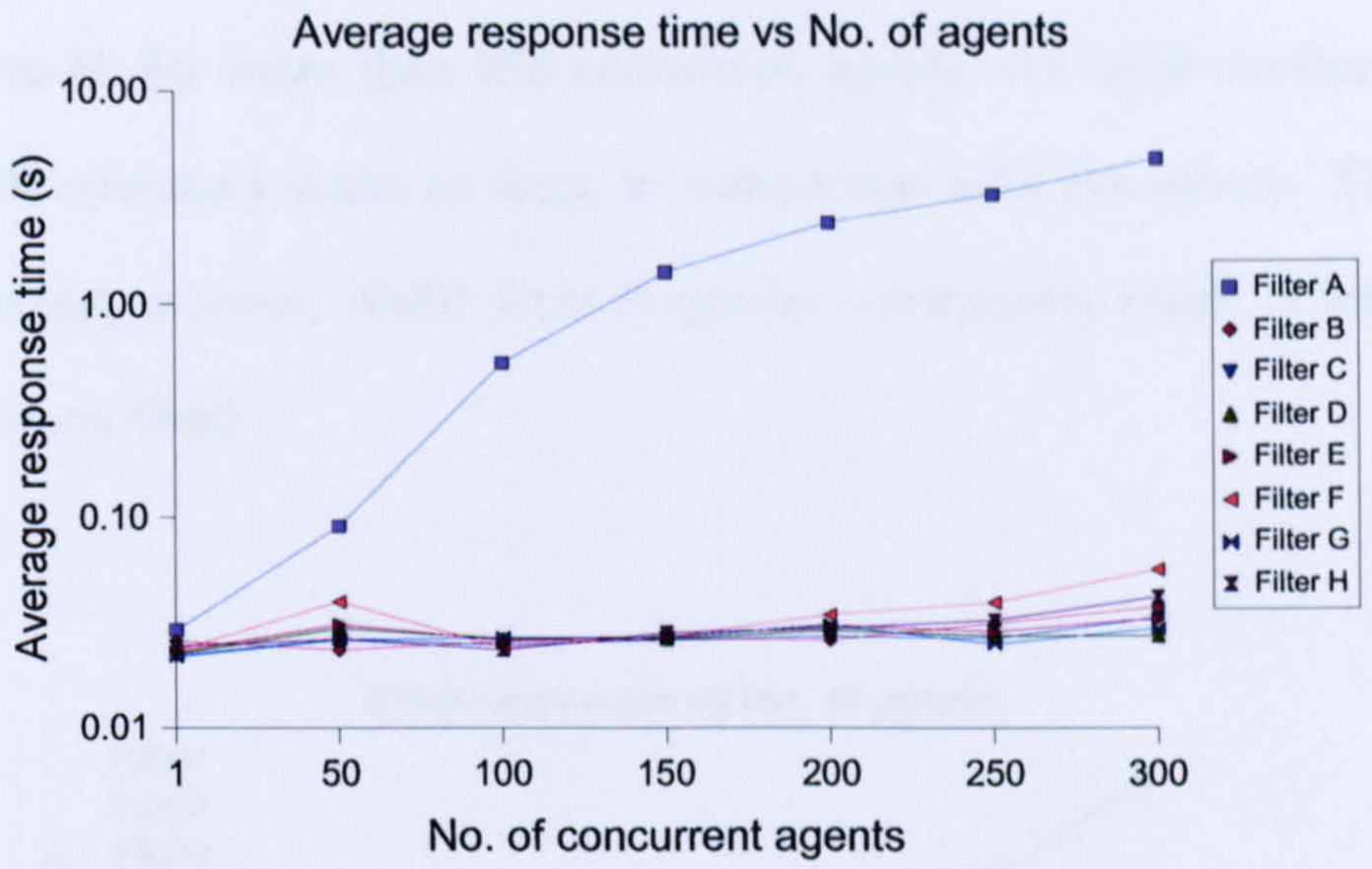

Figure 4.22: Comparison of average response time with disparate queries (EE).

the number of queries that can be processed per unit time. However, it can be observed that the throughput actually increases at around 100 concurrent agents, for filter H. This behaviour can be explained by the number of empty result sets due to the filter combination, and to caching in the GRIS. Additionally, the average throughput is more clearly differentiated at 300 agents, where filter D (the OR operator with two conditions) was the highest.

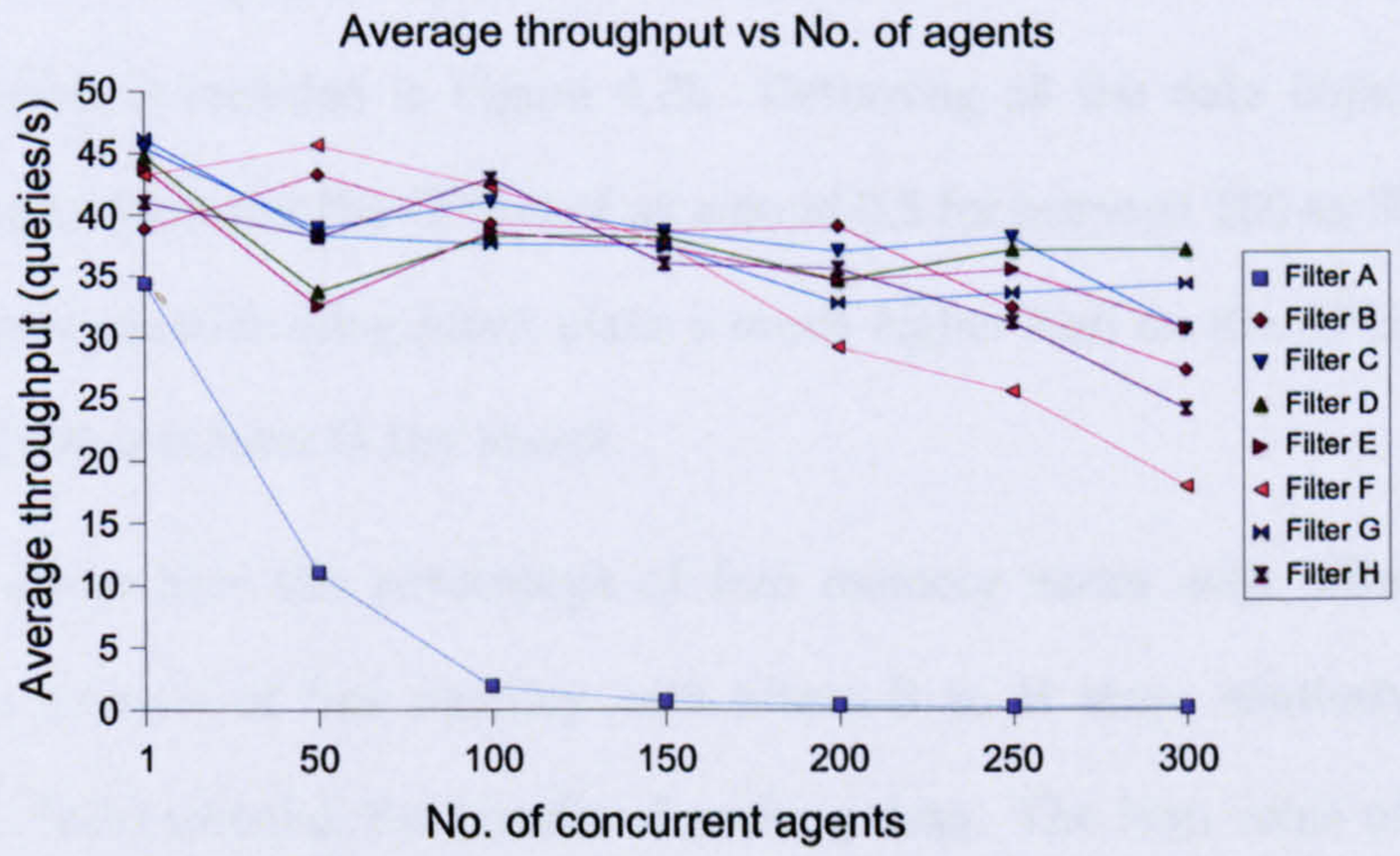

Figure 4.23: Comparison of average throughput with disparate queries (EE).

The total responses graph in Figure 4.24 shows that almost identical results are obtained with filters B to H. For more than 250 concurrent agents, the total number of responses for filter F (OR operator) starts to drop, in comparison with the others. The increase in responses received, is linear, whilst filter A queries consistently result in less than 15000 responses to be received.

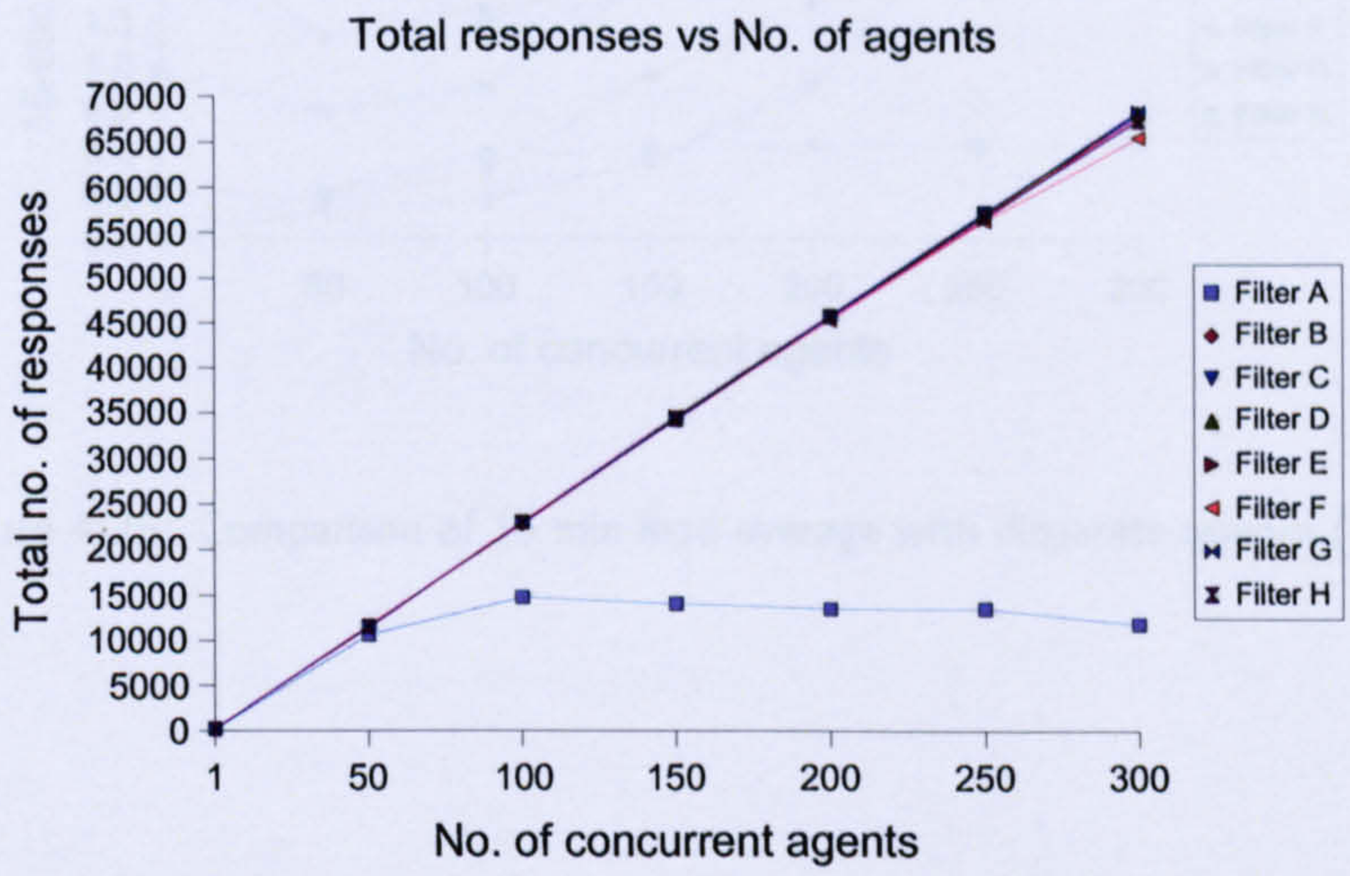

Figure 4.24: Comparison of total number of responses with disparate queries (EE).

Since the 15 min load graph shows more marked difference in filters than the 1 min load graph, the former is included in Figure 4.25. Returning all the data objects within the requested scope, maintains the CPU load at around 0.5 for between 100 to 300 concurrent agents. However, queries using filters place a much higher load on the CPU, with filter F being the highest and filter B the lowest.

Figure 4.26 shows how the percentage of free memory varies with different types of queries. The amount of free memory with filters B to H stays relatively constant at around 2.4%, demonstrating the benefit of caching data. The high value of free memory (about 3.8%) for filter A suggests the unavailability of data in memory at the beginning of the experiment.

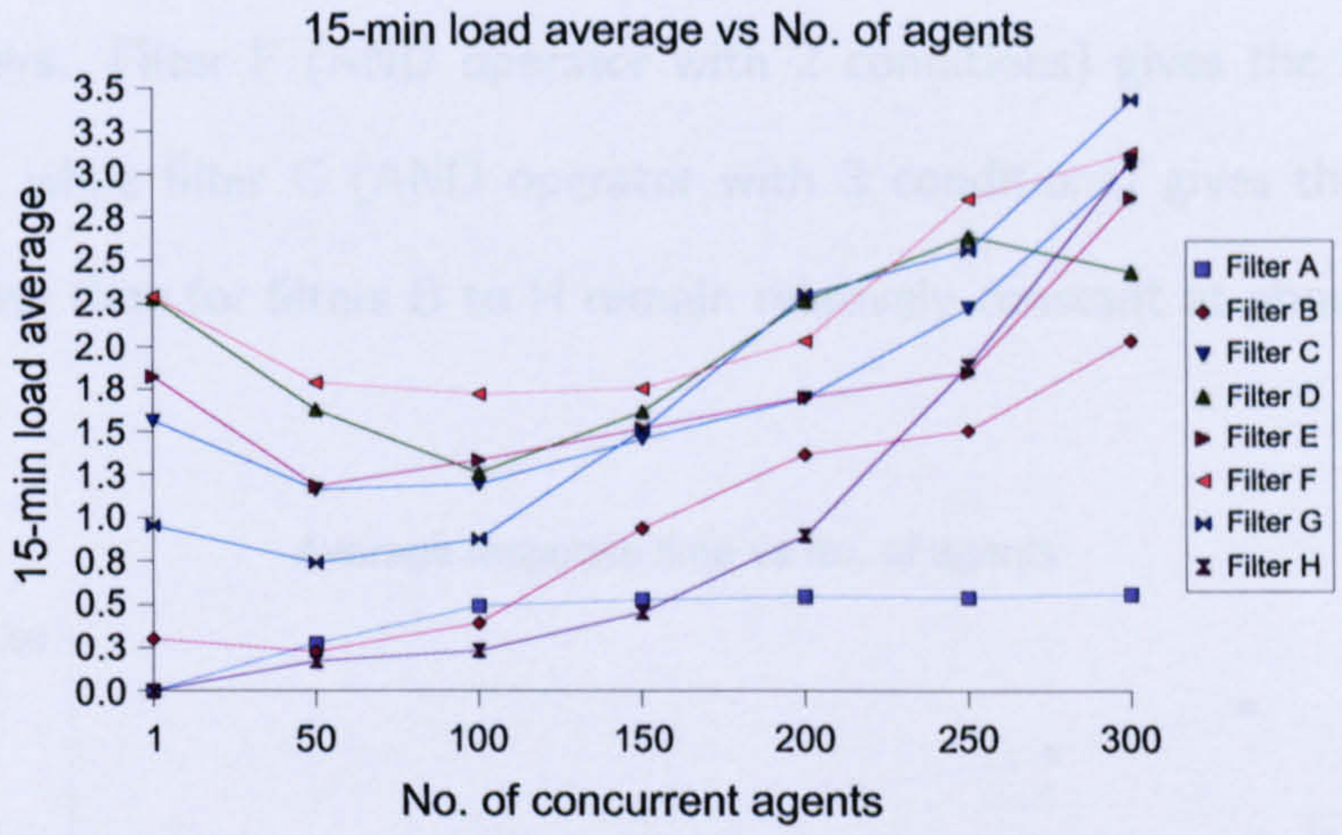

Figure 4.25: Comparison of 15 min load average with disparate queries (EE).

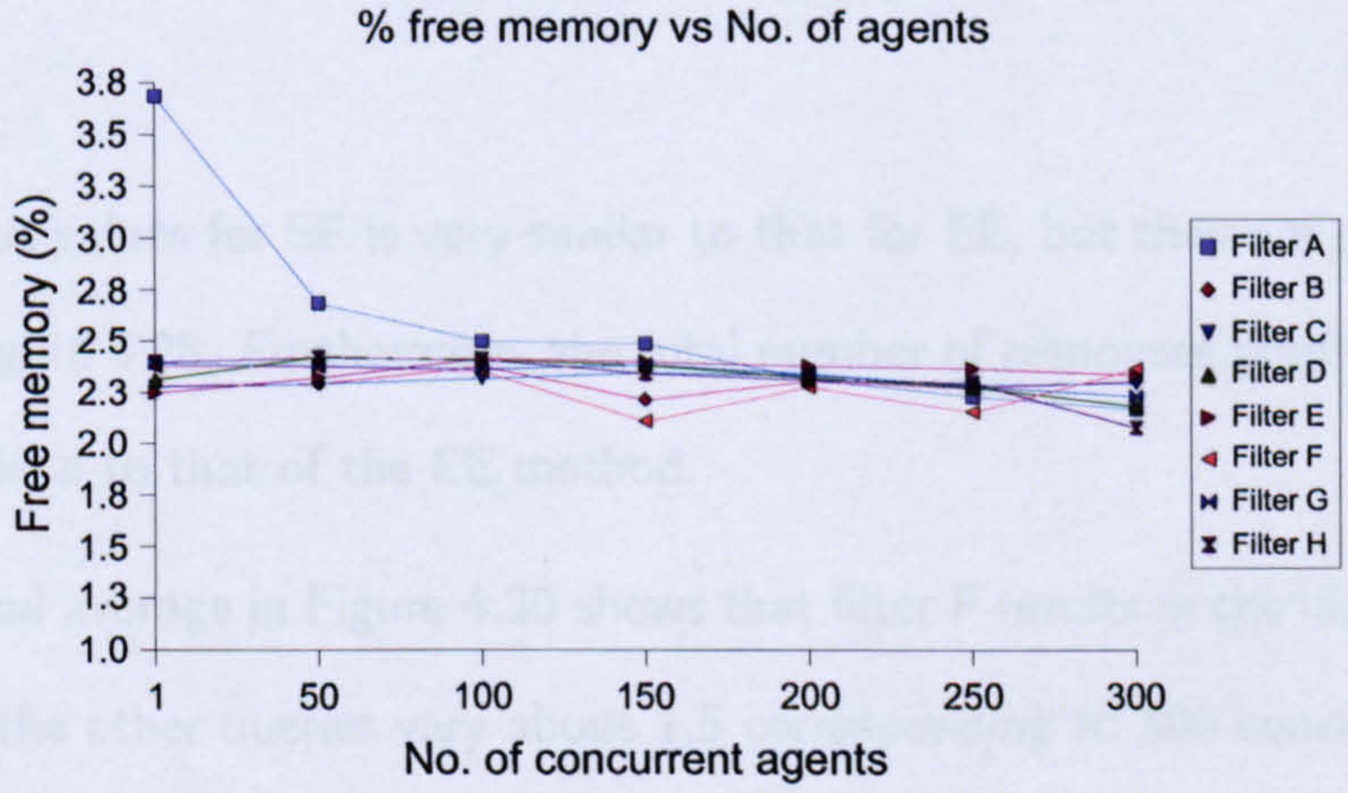

Figure 4.26: Comparison of percentage of free memory with disparate queries (EE).

Experiment Results with Speculative Evaluation

Figure 4.27 shows that the average response time with filter A is very different from that with other filters. Filter F (AND operator with 2 conditions) gives the highest overall response time, while filter G (AND operator with 3 conditions) gives the lowest. The average response time for filters B to H remain relatively constant at about 0.07 s.

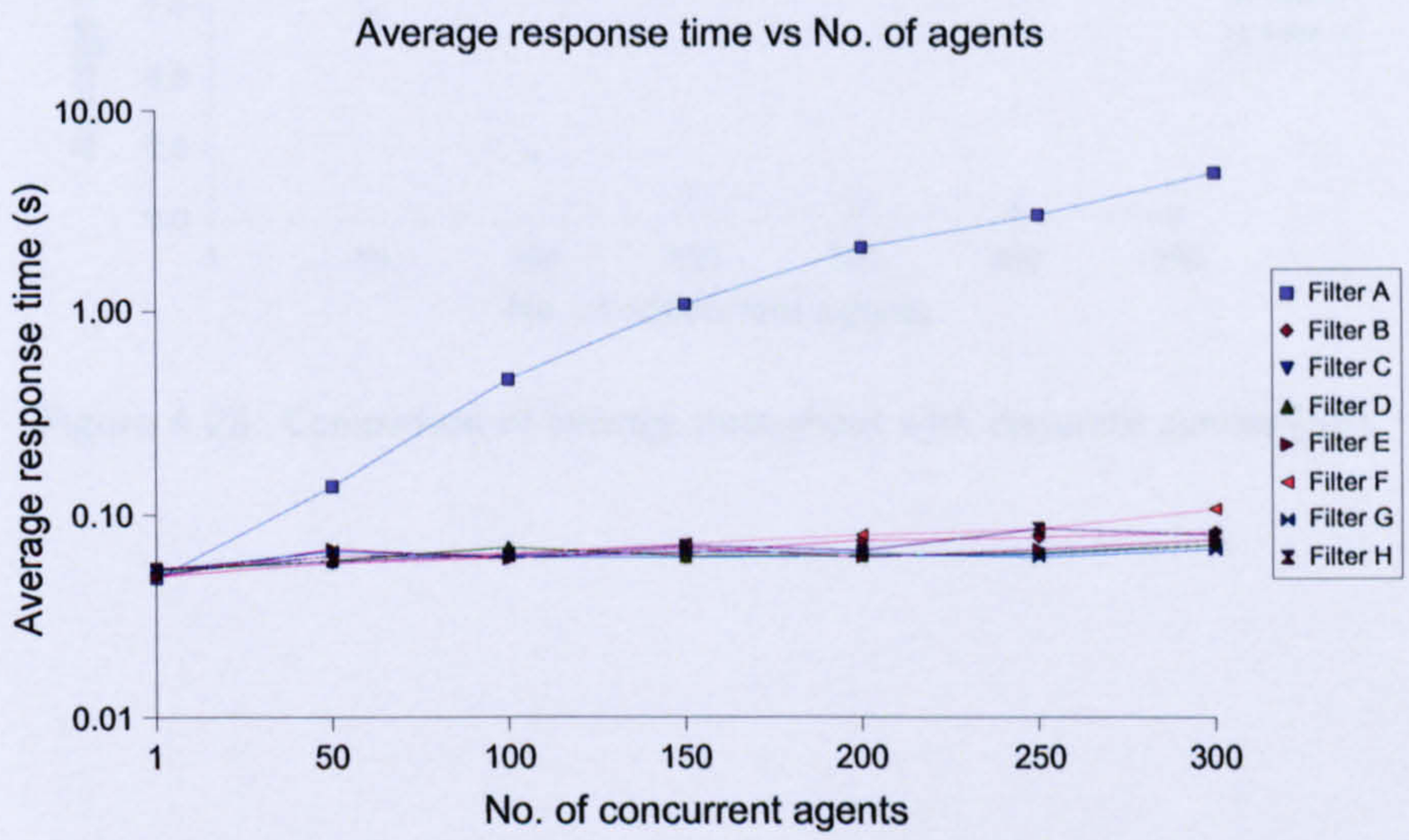

Figure 4.27: Comparison of average response time with disparate queries (SE).

The throughput values for SE is very similar to that for EE, but they are generally lower, as shown in Figure 4.28. Furthermore, the total number of responses graph in Figure 4.29 is almost identical to that of the EE method.

The 15 min load average in Figure 4.30 shows that filter F results in the highest load, and the load with the other queries vary above 1.5 corresponding to 300 concurrent agents.

There is a relatively small difference in the percentage of free memory across the various queries. Nevertheless, filter E has a sudden increase in free memory beyond 100 concurrent agents. This can be explained by the relatively smaller result set returned, due to the filter not being satisfied, as shown in Figure 4.31.

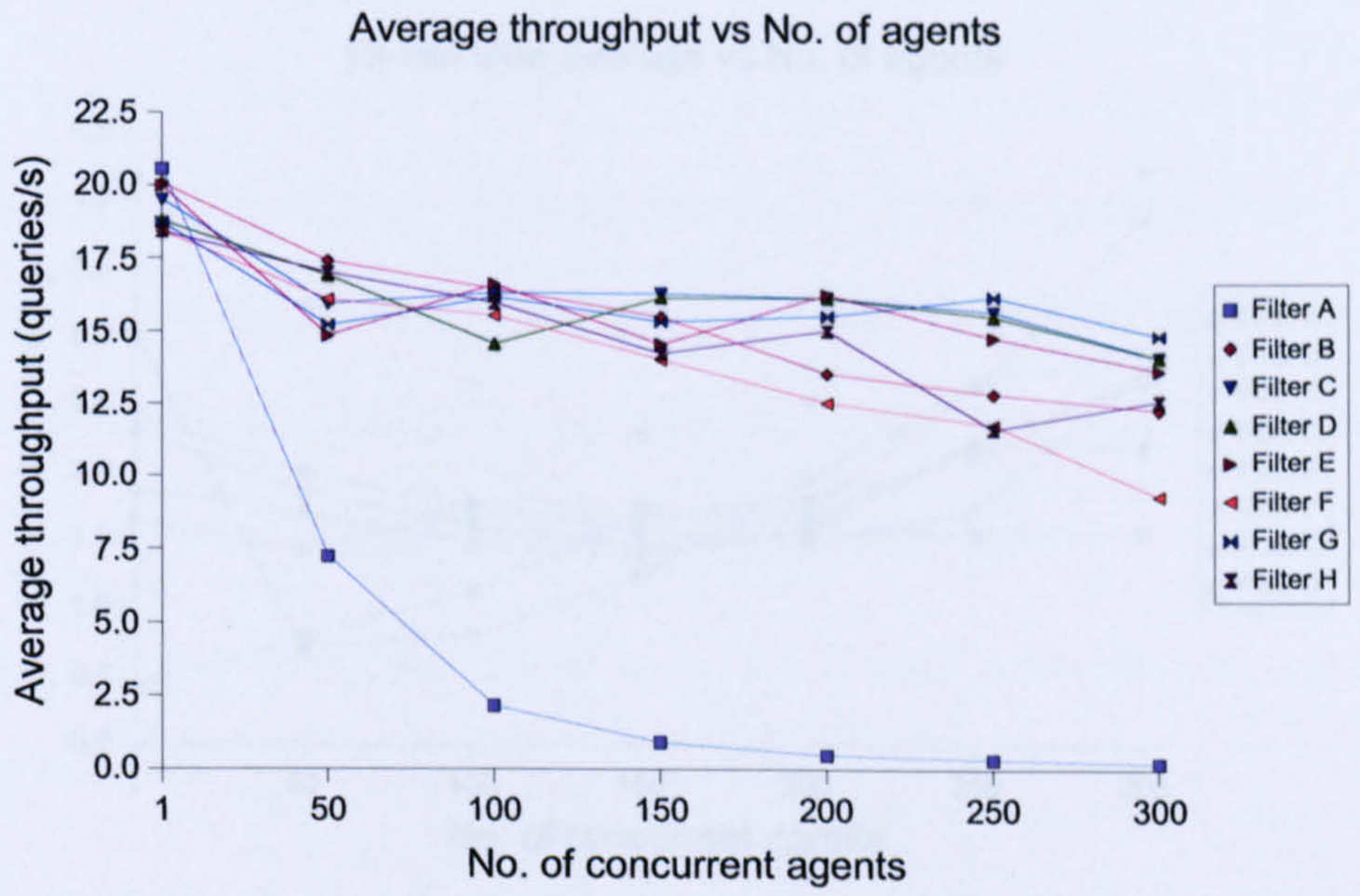

Figure 4.28: Comparison of average throughput with disparate queries (SE).

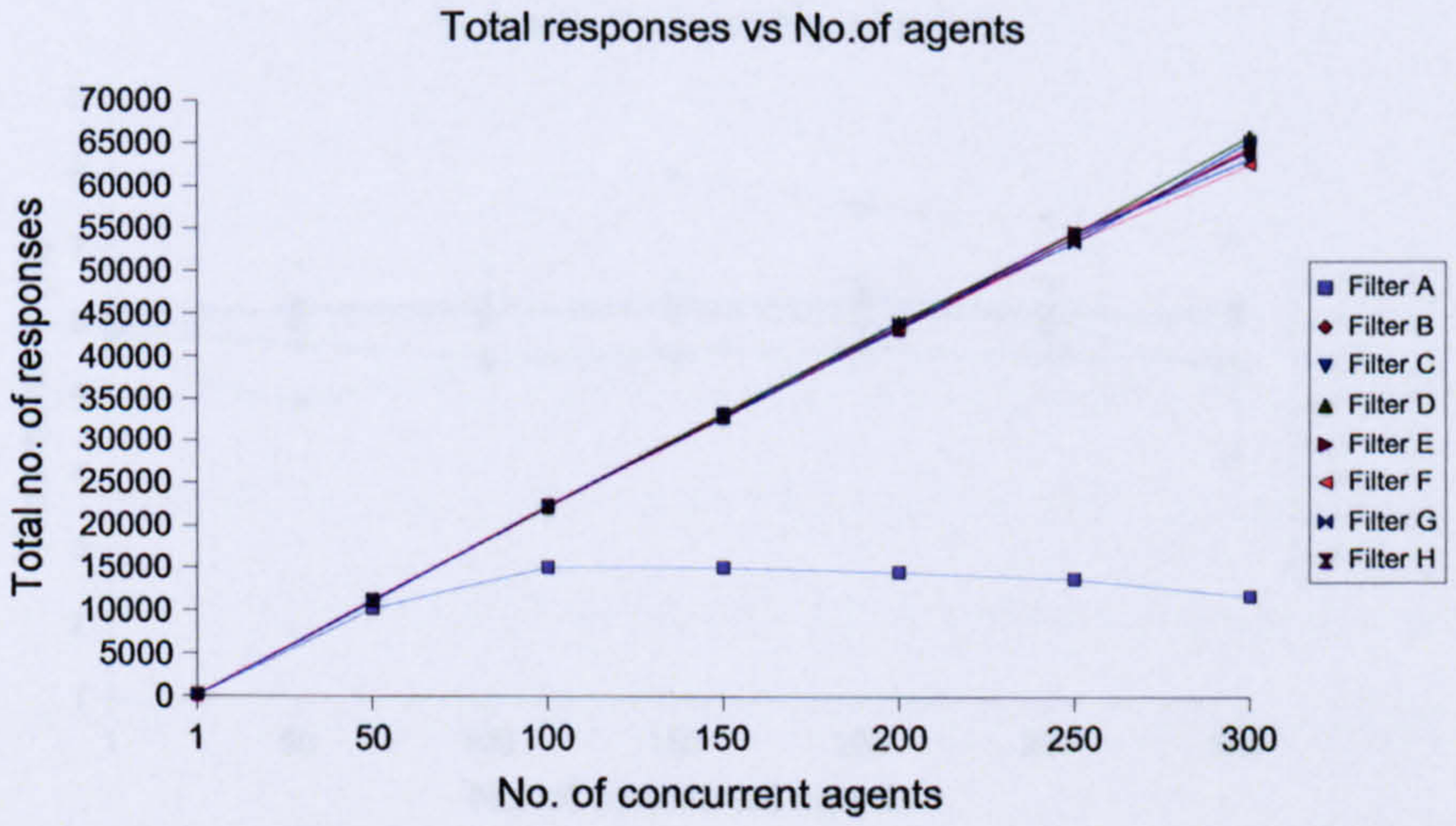

Figure 4.29: Comparison of total number of responses with disparate queries (SE).

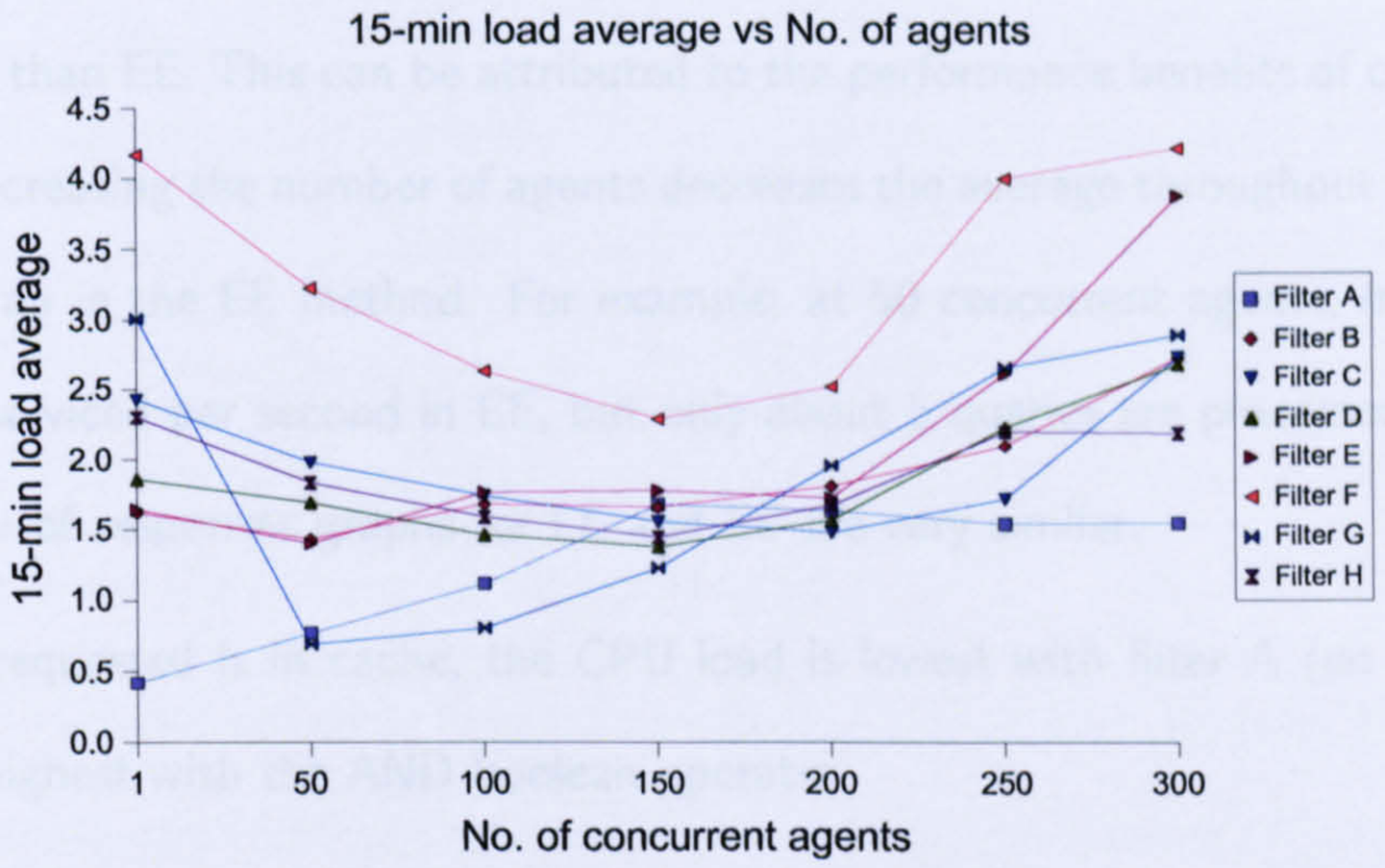

Figure 4.30: Comparison of 15 min load average with disparate queries (SE).

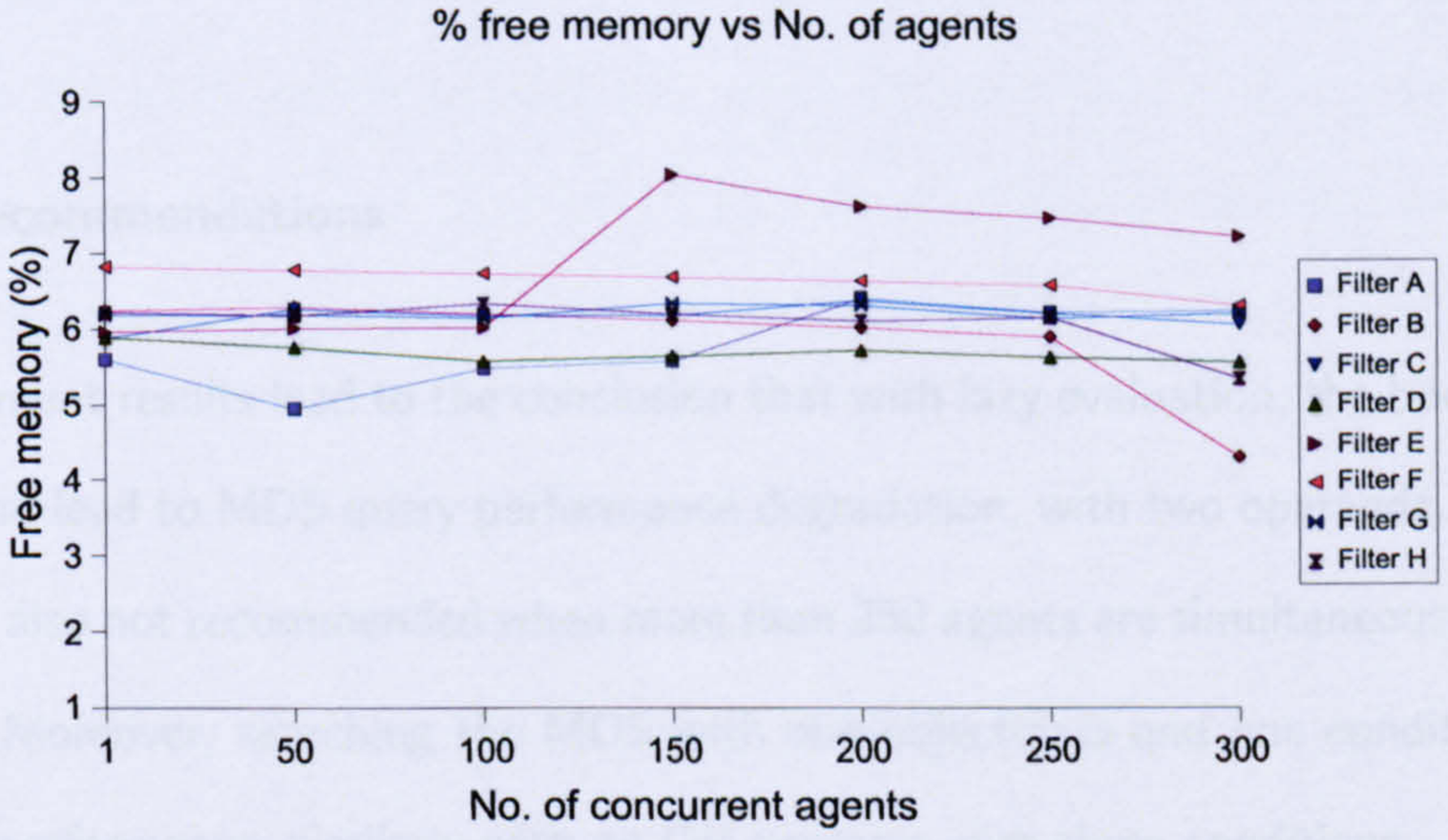

Figure 4.31: Comparison of percentage of free memory with disparate queries (SE).

Comparing Evaluation Methods

Queries with the NOT operator take longer to be processed when the evaluation method is LE, rather than EE. This can be attributed to the performance benefits of caching data. Moreover, increasing the number of agents decreases the average throughput more rapidly in the LE than in the EE method. For example, at 50 concurrent agents, more than 10 queries are serviced per second in EE, but only about 5 queries are processed in LE. The total number of responses graphs for LE and EE are very similar.

If the data requested is in cache, the CPU load is lowest with filter A (no filter on the data), and highest with the AND boolean operator.

The average response time graph for SE follows that for EE more than that for LE. This is because data is made available in the SE method, on a periodic basis for queries about dynamic data. Furthermore, the total number of responses handled within a given time period (10 min) is the same in the EE and SE methods.

Over the course of the experiments, the 15 min load average for the SE method was higher than that for EE. This is because of the database communication.

4.7.5 Recommendations

The experiment results lead to the conclusion that with lazy evaluation, the boolean AND operator can lead to MDS query performance degradation, with two operands. The NOT operator is also not recommended when more than 250 agents are simultaneously querying the MDS. Moreover, searching the MDS with one objectclass and one condition results in reliable performance; similarly with an OR operator with three conditions.

Query filtering reduces the average response time because certain parts of the LDAP DIT (Directory Information Tree) are not traversed at all. While specifying exact searches can reduce the average response time, certain operators can perform poorly but never worse

than without filtering.

If the aim of the MDS is to maximise the total number of queries that it can process over a given period of time, searches with the AND and OR boolean operators should be limited. For about 100 agents, queries with the OR operator and only one condition give the least CPU load. Moreover, if the amount of free memory is important, then queries with the OR operator and those returning all data objects should be limited. More memory allows the MDS to process queries in less time, but an increasing number of AND and NOT filters can considerably increase the response time.

Therefore, to optimise the performance that clients perceive from querying the MDS with disparate queries, the MDS should control and reduce the number of queries with the AND and the NOT operators. However, it is also observed that the average response time can partly depend on whether the filter condition is satisfied or not. In order to improve performance, complex queries involving the boolean AND operator could be split into several simpler queries and the results further filtered on the client side. Alternatively, complex queries could be rewritten but made to retain their expressive power.

Searching the MDS with filters also allows the performance prediction of certain metrics, for example the total number of responses which increases linearly for complex queries. Additionally, such queries can significantly increase the resource overhead on the MDS host, thereby requiring adequate machines to act as the GIIS.

Chapter 5

MDS3 Benchmarking

5.1 Introduction

The focus of this chapter is the provision of very dynamic data about Grid resources. The performance of the Index Service with push and pull queries is studied, with an increasing number of users. Notification sources allow the propagation of changes occurring in the resources being monitored, to sinks which have shown an interest in these resources. This chapter first details the benchmarks and subsequently, analyses the performance observed at the client of the MDS3 Index Service when different notification (push-based) rates are utilised. Additionally, the overhead at the Index Service is investigated as the number of concurrent clients scales. The approach taken is to ensure that a data object is never out-of-date by more than a given number of time units with its source. That refresh rate determines the nature of the provided guarantee - the larger its value, the weaker the consistency guarantee; for instance, the data object could be out of date by up to that number of time units at any point. This method of using data consistency provides a quantitative, characterisable guarantee by means of an upper bound on the amount by which a data object could be stale.

This method has the advantage of setting a consistency margin on very dynamic data which has the impracticality of having a notification sent to a sink each time its value changes. This mechanism thus diminishes the possibility of causing communication bot-

tlenecks at the server. This chapter also details the performance prediction of push-based Grid Information Systems, based on the performance data benchmarked.

Grid environments create the implicit need for applications to obtain real-time information about the meta-system's structure and state, to utilise this information to make configuration decisions, and to be notified when information changes. This chapter and the following one, both address these issues, drawing on the more recent Grid Information Service which is the OGSA-based (Open Grid Services Architecture) [35] MDS3. The contribution made by this chapter is therefore the evaluation and characterisation of the performance achievable by the Globus Monitoring and Discovery System (MDS3) which is a widely deployed reference implementation of a Grid information service. The context of this research work is the delivery of dynamic, up-to-date data to clients using the MDS3 push-based mechanism. Prediction performance techniques are also proposed to characterise accurately the behaviour of the MDS in supporting a number of concurrent notification sinks. Performance prediction allows Grid applications to make decisions to minimise the overhead incurred whilst ensuring QoS enforcement for the clients.

With the emergence of the Grid, distributed applications have more stringent QoS requirements which can be mapped to low-level resource requirements. More specifically, those system QoS requirements can be applied to resource discovery and monitoring services. Understanding how the MDS3 clients can individually and as a whole, affect its performance, is crucial to ensure that applications meet their QoS requirements. The current GIS state information is then utilised in dynamic, adaptive algorithms to maintain the level of service to clients. Additionally, resources making up a virtual organisation (VO) can be very dynamic and therefore can change state as often as every few seconds. The essence of the proposed work is the use of a reasonably accurate approximation of the current resource status and availability. This approximation is provided by a notification relationship between the resources'mediator (the MDS) and clients, where the notification rate can vary to match the rate of change in the resources.

Furthermore, whilst network monitoring is a component of the whole resource discovery process, this chapter concentrates on the characterisation of the behaviour of the MDS alone, without the external influence of network conditions. The focus is thus on the performance of the actual MDS push-based mechanism solely, which is best done and examined using a LAN. This research work is supported in [70].

The second part of this chapter presents the scalability and performance studies of the Index Service with an increasing number of pull-based queries. The effect on the MDS of different wait times between consecutive queries as well as various cache TTL values is analysed.

5.2 Push-based Benchmarks

The details of how the experiments were performed, now follow. The experiment results are then evaluated, based on the performance metrics chosen.

5.2.1 Experimental Setup

The experiments were carried out on a Grid testbed, which had the Globus Toolkit 3 installation. The version of the GIS utilised was MDS3 since GT3 was the current latest version of the Grid middleware software for the purpose of the research.

Across the various experiments that have been carried out, an agent would represent a Grid application and act as a notification sink. The agent was written using the OGSA 3.0.2 APIs [47] and it subscribed to the Index Service for changes in one of its service data elements (SDEs). These notification sinks were set up on a maximum of ten machines ($\mathcal{M}_1$ to $\mathcal{M}_{10}$). With a maximum of 500 agents simultaneously receiving notifications from an Index Service over a period of five minutes, the objective was to load-balance the notification sinks and to stress-test the notification source. The maximum number of agents attributed to one machine was therefore 50 (same set-up as in the first footnote

in Section 4.5.3). In these experiments, the client machines are connected to the Index Service host. The number of notifications each agent received within the duration of the experiment was monitored. Furthermore, several characteristics were observed on the Index Service, namely its throughput, CPU load and memory utilisation.

As for the set up of the Index Service, it was configured on a Linux kernel 2.4.18-14 machine with a 1.9 GHz processor and 512 MB RAM. Moreover, the Index Service was running in a Tomcat container, version 4.1.27, on port 8080.

In short, during the experiments, the performance of the Index Service was monitored as it received an increasing number of subscriptions. Performance data was collected from a set of ten experiments, and the average results are shown in the following sub-section.

5.2.2 Subscription and Notification Setup

The configuration of the Index Service offers an extensible framework for the management of dynamic Grid data. The above experimental setup makes use of the interface provided for connecting external service data provider programs to service instances. These external provider programs can either be the core providers that are part of GT3 or user-created ones. The core *SimpleSystemInformation* provider is used in the experiments.

Another interface of the Index Service is also used, namely the *NotificationSource* interface which is used for client subscription. An agent is written as a client which registers interest in changes in the Index Service. Notification messages are sent about changes in the service data element created by the *SimpleSystemInformation* provider. The rate at which that service data element is created is varied during the experiments to observe the effect on the Index Service overhead and the performance at the notification sinks. Moreover, the agent implements the *NotificationSink* interface which allows the asynchronous delivery of notification messages. Therefore, this enables the dynamic discovery and monitoring of Grid resources.

An agent is set up so as to be subscribed to the Index Service for the *Host* Service Data Element produced by the *SimpleSystemInformation* provider. When the *SimpleSystemInformation* provider is configured to run every 60 seconds, the client receives notifications every minute.

5.2.3 Performance Metrics

The following performance metrics are used to assess the performance of the Index Service in sending notification messages to its clients:

- 1 min CPU load average ($\mathcal{L}_1$). The load average is indicative of the Index Service being under heavy usage, which increases the average response time and time-outs.

- 5 min CPU load average ($\mathcal{L}_5$);

- Percentage of CPU idleness ($\mathcal{CPU}_{idle}$). CPU utilisation is a function of the load on the Index Service, and a high value indicates that the Index Service host can no longer support more notifications and performance will therefore drop.

- Percentage of memory utilised ($\mathcal{M}_{util}$). The required memory increases with the number of sink connections and is indicative of the usage of the Index Service.

- Notification throughput ($\mathcal{T}$). This is a server-side metric (number of notifications sent per second) which demonstrates whether the Index Service scales with an increasing number of concurrent queries.

- Percentage of agents receiving x% of notifications ($\mathcal{N}_x$). With more sinks or at a higher rate of notification, the performance of the Index Service starts to drop, and not all agents receive their expected rate of notification. This metric shows the percentage of notifications at the expected rate.

5.2.4 Experiment Results and Evaluation

The experiments examine the scalability of the Index Service with an increase in the number of notification sinks registered to it.

Figure 5.1 shows the change in the 1 min load average as the number of sinks increases; the load average is measured as the load on the Index Service during the last minute. The load average is therefore a measure of the number of jobs waiting in the run queue. It is observed that the highest load average occurs when the notification rate is 1 s. The load average peaks to just under 10.00, corresponding to 100 notification sinks, but it gradually decreases slightly for more sinks. Such observations are seen because the smallest value for the notification rate places the most load on the Index Service, and that at most 100 notification sinks can simultaneously be sent messages at the rate of once every second. On average, the load is proportional to the notification rate. However, it is observed that when there is only one notification sink, a notification rate of 30 s gives the lowest 1 min load average of 0.026; the rest of the notification rates range from about 0.06 to 0.1. Moreover, the 10 s notification rate results in the most variable load average increase over time, while the 30 s rate gives the most consistent rise in load average.

Figure 5.2 shows the corresponding 5 min load average graph which is consistent with the 1 min load average. Here again, the 1 s notification rate gives the highest average load, and the 5 min notification rate the lowest. Furthermore, for all the notification rates except for 1 s and 10 s, $\mathcal{L}_5$ decreases as the number of sinks increases from 1 to 25. This can be explained by the initial system overhead at the start of the experiment.

The change in the percentage of memory utilised is shown in Figure 5.3. As expected, the 1 s notification rate utilises the most memory, but as the number of notification sinks increases, $\mathcal{M}_{util}$ follows an downward trend. There is around a 15% decrease in utilised memory with 500 simultaneous sinks. This indicates that memory usage is released with more notification sinks. Nevertheless, this is not the case for the other notification rates

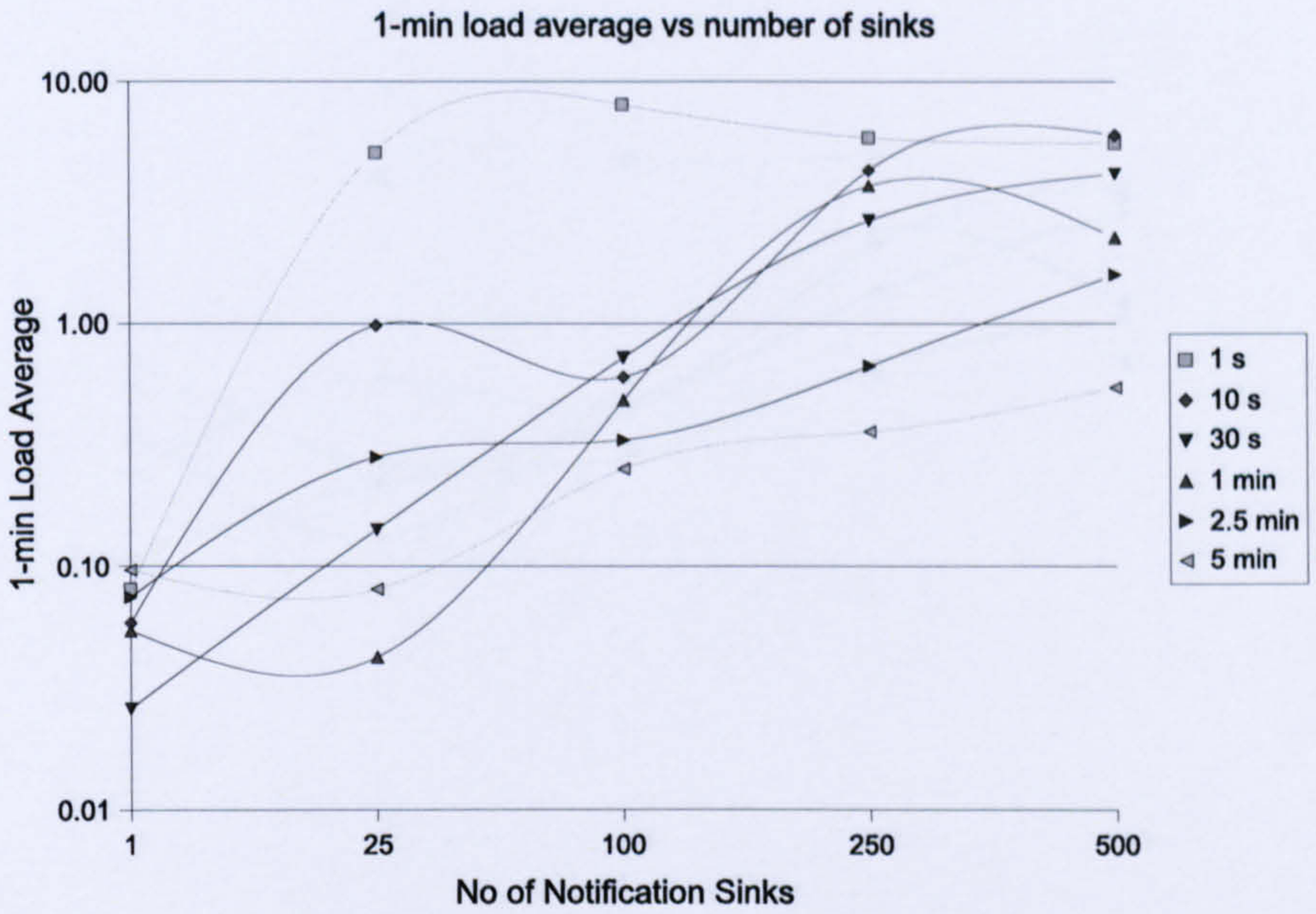

Figure 5.1: Experiment: 1 min load average.

where $\mathcal{M}_{util}$ has a general upward trend. In general, the greater the notification rate, the less memory being used.

Figure 5.4 shows how the percentage of CPU idleness varies with an increase in the number of concurrent sinks. On average, the 1 s notification rate uses the most CPU processing power. As the number of notification sinks increase beyond 1, the CPU idleness decreases, but for more than about 25 concurrent sinks, the CPU idleness starts to increase reaching around 60% for 500 sinks. All the other notification rates produce a value of around 97% for 1 notification sink, but $\mathcal{CPU}_{idle}$ has a downward trend with more notification sinks. This trend is explained by the overhead placed in the CPU with more sinks. It is also observed that for all the notification rates, apart from 1 s and 10 s, $\mathcal{CPU}_{idle}$ stays relatively stable for under 25 concurrent sinks. However, for more than 25 sinks, the difference in CPU idleness starts to be apparent.

The throughput performance metric is a measure of the number of notifications which the Index Service sends out on average every second. The experiment results showing $\mathcal{T}$,

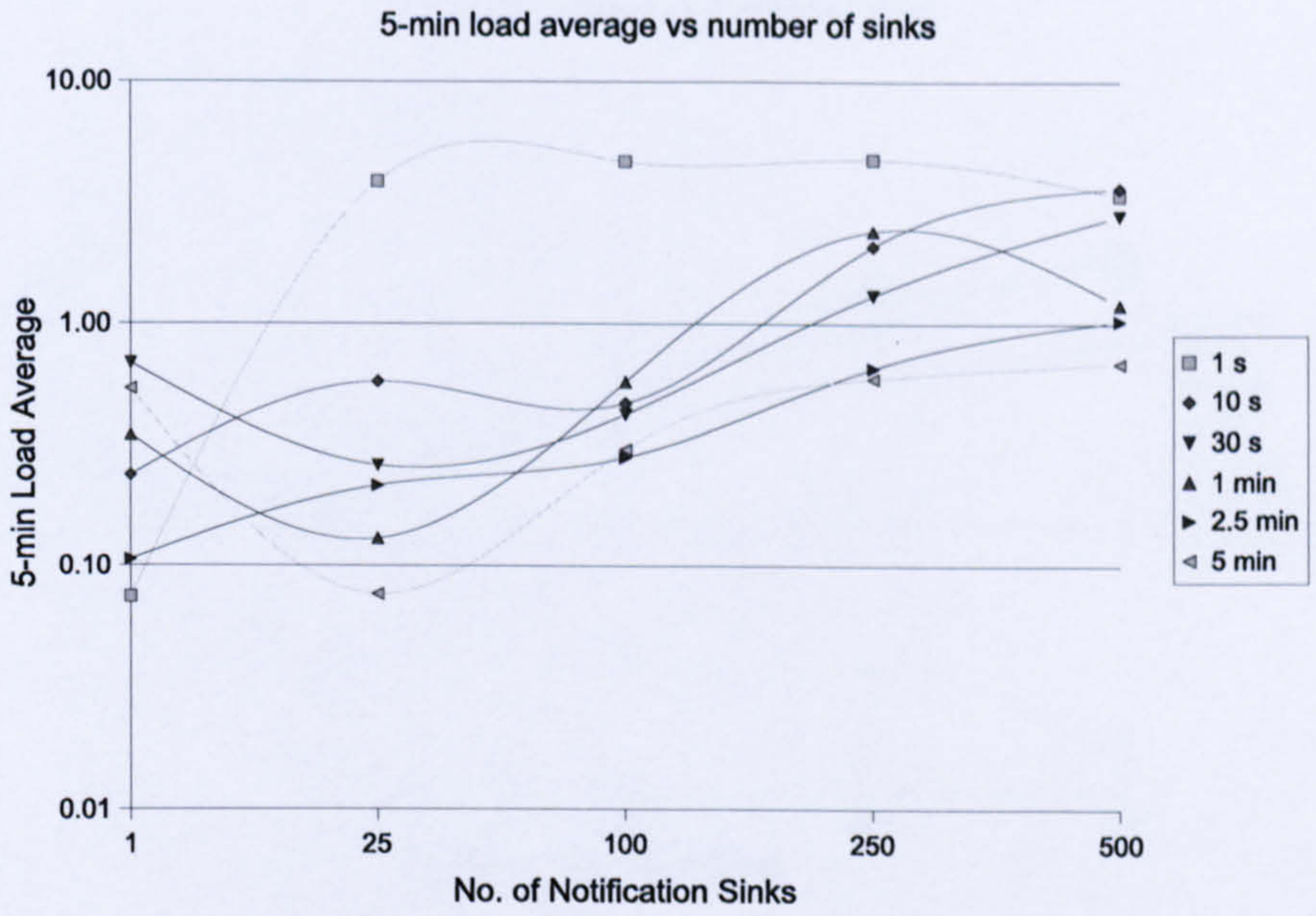

Figure 5.2: Experiment: 5 min load average.

is given in Figure 5.5. All the curves, except for the 1 s and the 10 s ones, show a slight quadratic increase in the throughput. The difference in $\mathcal{T}$ with these notification rates are rather distinctive. Nevertheless, the 1 s and 10 s notification rates results both indicate a peak in $\mathcal{T}$. This occurs at just over 25 concurrent sinks with the 1 s notification rate, and at just over 250 sinks for the 10 s rate. These results suggest that there is a maximum value for the throughput and hence, the number of concurrent notifications which the Index Service can deliver per unit time. For the experiments performed, this value is just under 25 notifications per second.

The number of notifications each agent should receive throughout the length of the 5 min experiment varies, depending on the number of concurrent notification sinks and the notification rate. On occasions, each agent can receive all scheduled notifications; on others, it receives a smaller number of messages. Figures 5.6 and 5.7 show the percentage of agents receiving 100%, 90%, 70% and below 70% of scheduled notifications when the notification rates are 10 s and 30 s respectively. These results are chosen amongst the other notification rates because they show the most variability. As can be seen in Figure 5.6,

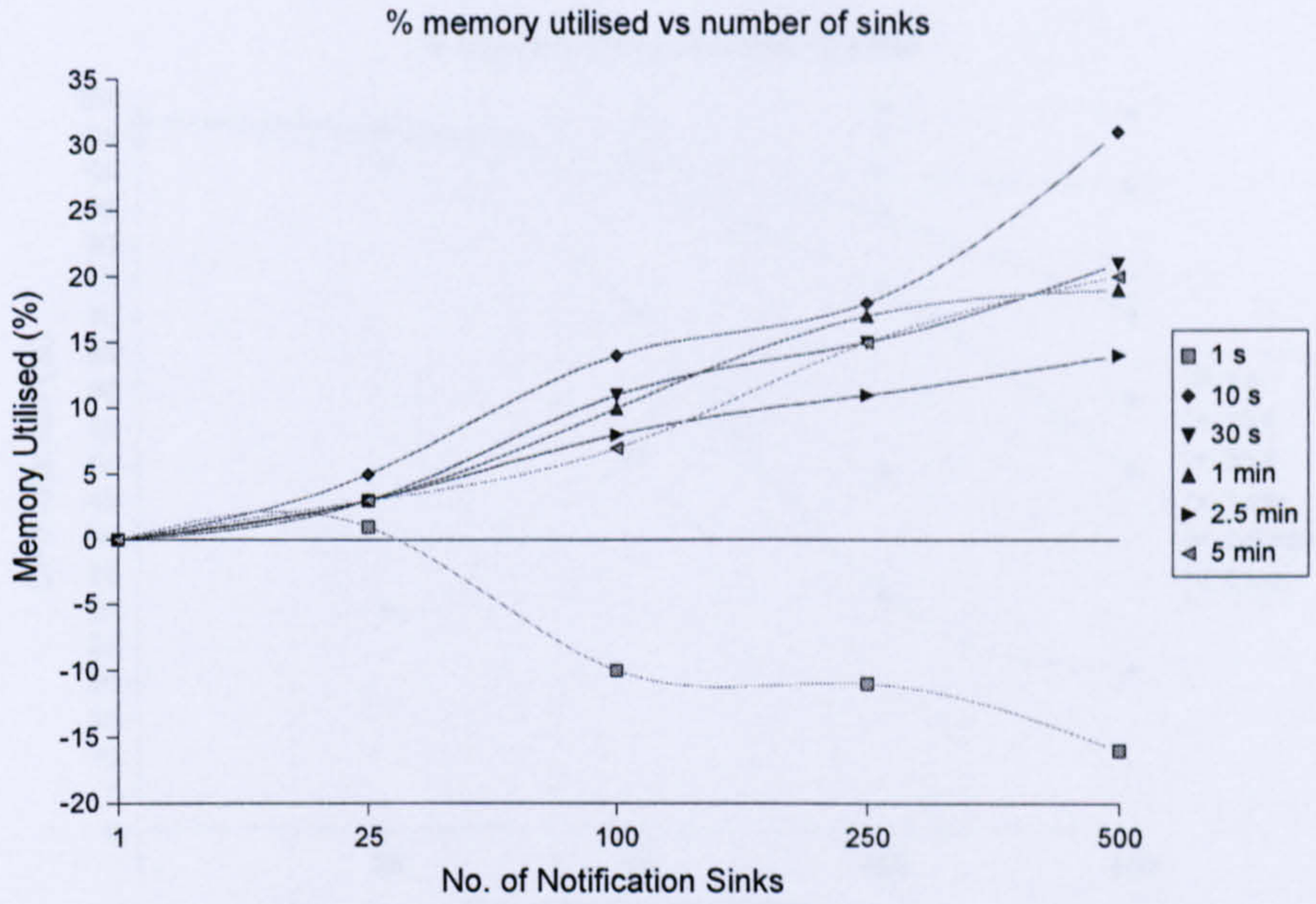

Figure 5.3: Experiment: percentage memory utilised.

the number of expected notifications were received by every agent, when the number of concurrent sinks is 25 or less. Moreover, when the number of concurrent sinks is 100, 80% of the agents on average receive all the notifications, and 20% of them receive 90% of the expected number of notifications. Similarly, when the number of notification sinks is 250, only 23.6% of the agents receive all notifications, while the rest of them receive 90% of the notifications. Additionally, the QoS experienced by the agents drop significantly with 500 concurrent notification sinks where all the agents receive less than 70% of the maximum number of notifications.

Similarly, Figure 5.7 shows that with a 30 s rate of notification, all the agents receive all the notifications expected when the number of concurrent sinks is less than 25. Moreover, 21.6% of the agents receive 90% of the notifications, while the remaining of the agents receive all the notifications. This behaviour occurs at 100 concurrent notification sinks. The behaviour at 500 concurrent sinks is more variable; 61.6% of the agents receive 100% of the notifications, 34% receive 90% of the notifications and 4.4% receive 70% of the notifications.

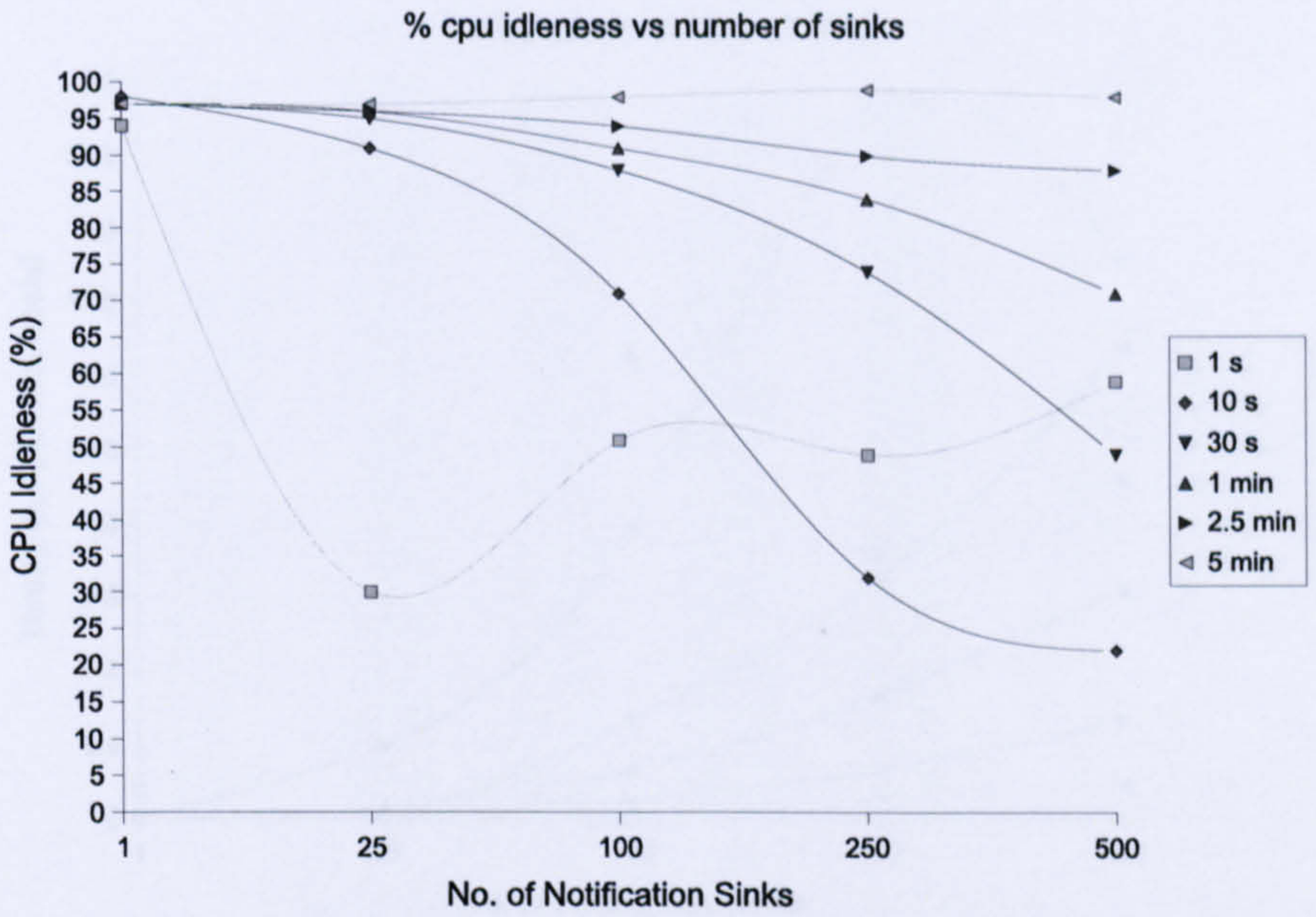

Figure 5.4: Experiment: percentage CPU idle.

The percentage of notifications received for the other notification rates are more stable. For example, all the agents receive all the notifications with a 1 s notification rate with less than or equal to 25 concurrent sinks. For more than 25 concurrent sinks, the agents receive below 70% of the total number of notifications. Furthermore, when the notification rate is 1 min or more, experiment results show that all the agents receive 100% of the notifications, unless the number of concurrent sinks is 500. Consequently, it can be observed that data consistency can suffer if the number of registered, active sinks increase beyond a certain value.

5.2.5 Push-based Benchmarking Summary

The approach taken by this section is to predict the behaviour of the Index Service from a Grid application's point of view, based on performance data benchmarks. These predictions are used to optimise the performance of the Index Service in handling a large number of concurrent notification sinks. This optimisation further contributes to guarantee quality-of-service in the use of Grid middleware. To do so, benchmarks of performance

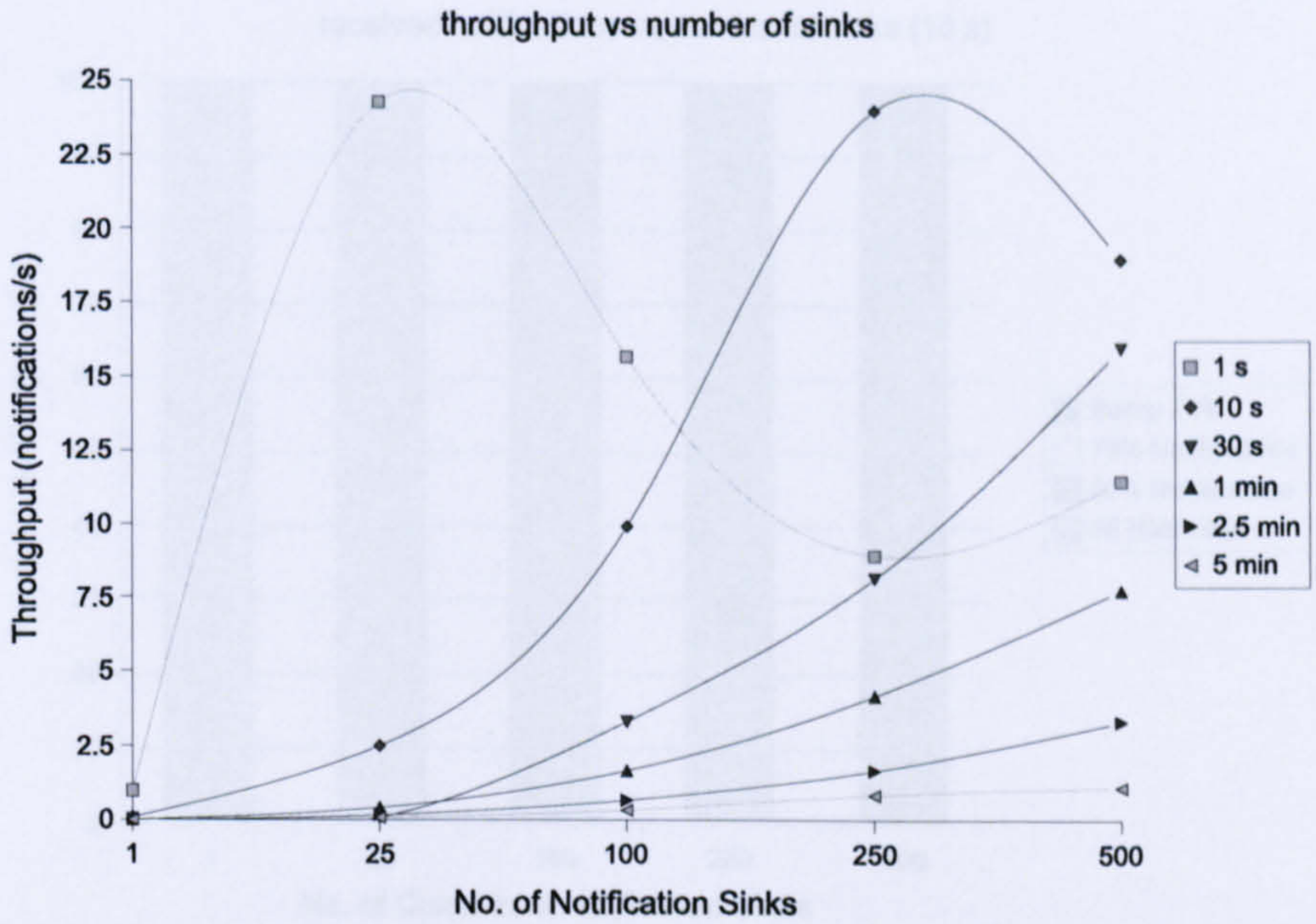

Figure 5.5: Experiment: throughput.

data is gathered about the behaviour of the Index Service as the number of concurrent notification sinks scales. Several scenarios have been set up where the notification rate varies. This rate provides a consistency guarantee of the data being monitored. Moreover, the experiment results demonstrate that a smaller value for the notification rate guarantees data consistency but at a higher overhead cost. It has also been found that such consistency cannot be maintained when the number of sinks increases.

5.3 Pull-based Benchmarks

The OGSI specification supports queries on service data, which is a powerful and complex functionality. Every OGSI-based Grid service exposes a `findServiceData` operation which enables clients to query the service for service data. The framework supports a default query which is based on the service data name *queryByServiceDataName*. Furthermore, the OGSI specification offers an extensible query capability which supports different query mechanisms. Several of the most popular queries are those by XPath, XQuery and

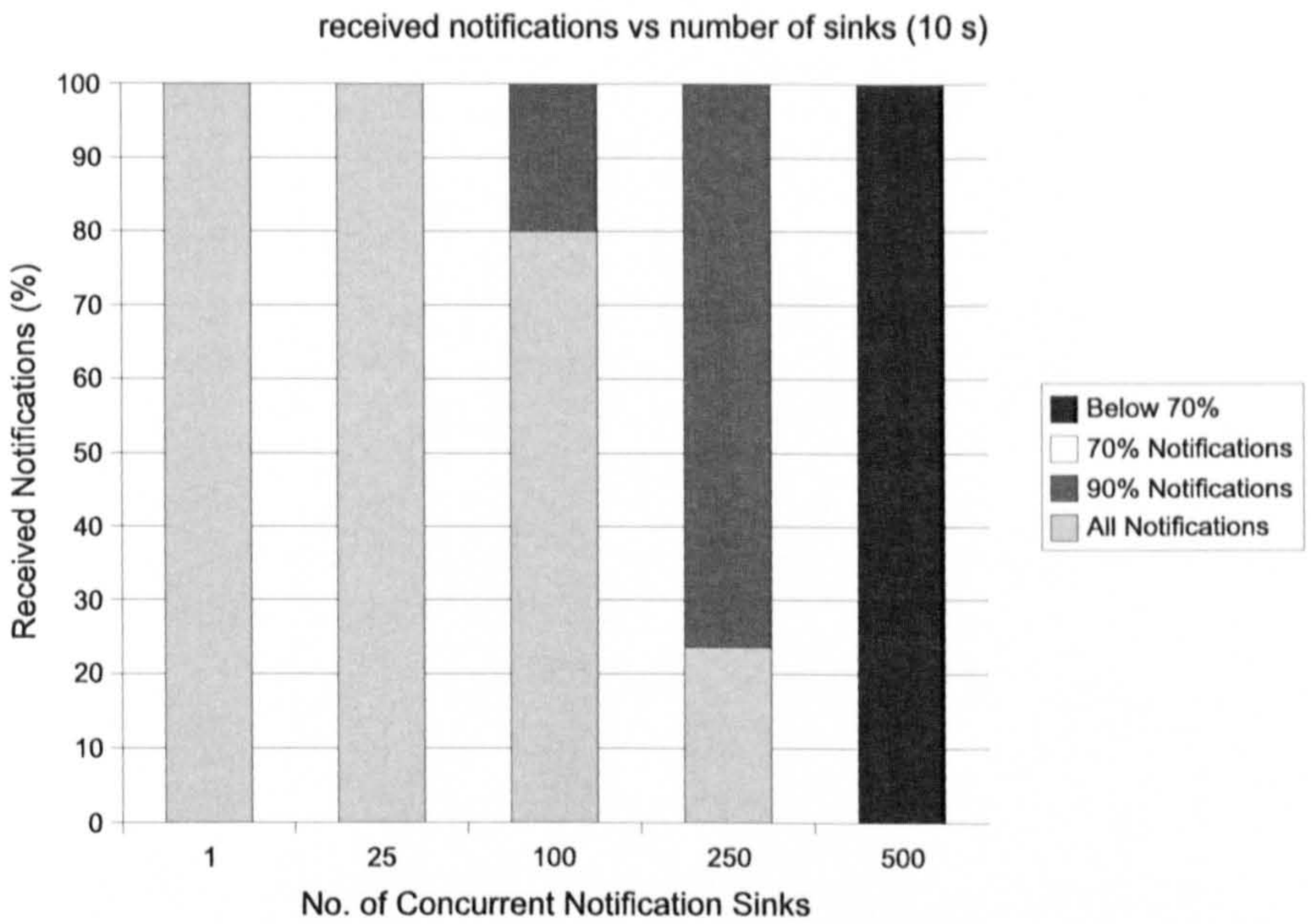

Figure 5.6: Experiment: percentage of received notifications (10 s).

SQL. Additionally, GT3 provides facilities for extensible query engine support.

This section describes a performance study of pull-based queries on the Index Service. For experimental consistency, pull-based queries will be carried out on the *SimpleSystem-Information* provider as in the push-based benchmarking experiments. This is a default Java-based host information provider available in the Index Service, and it produces data including the CPU count, memory statistics, operating system type and logical disk volumes.

5.3.1 Experimental Setup

These sets of experiments have a similar system setup as in the previous sets (Section 5.2.1). Throughout the first set of experiments, the refresh rate of the *SimpleSystemInformation* data provider is set to 1, 5, 30, 60 and 180 s, to analyse the effect of service data caching on the performance of the Index Service when handling synchronous queries. In these cases, the Index Service caches the SDE from the subscribed data

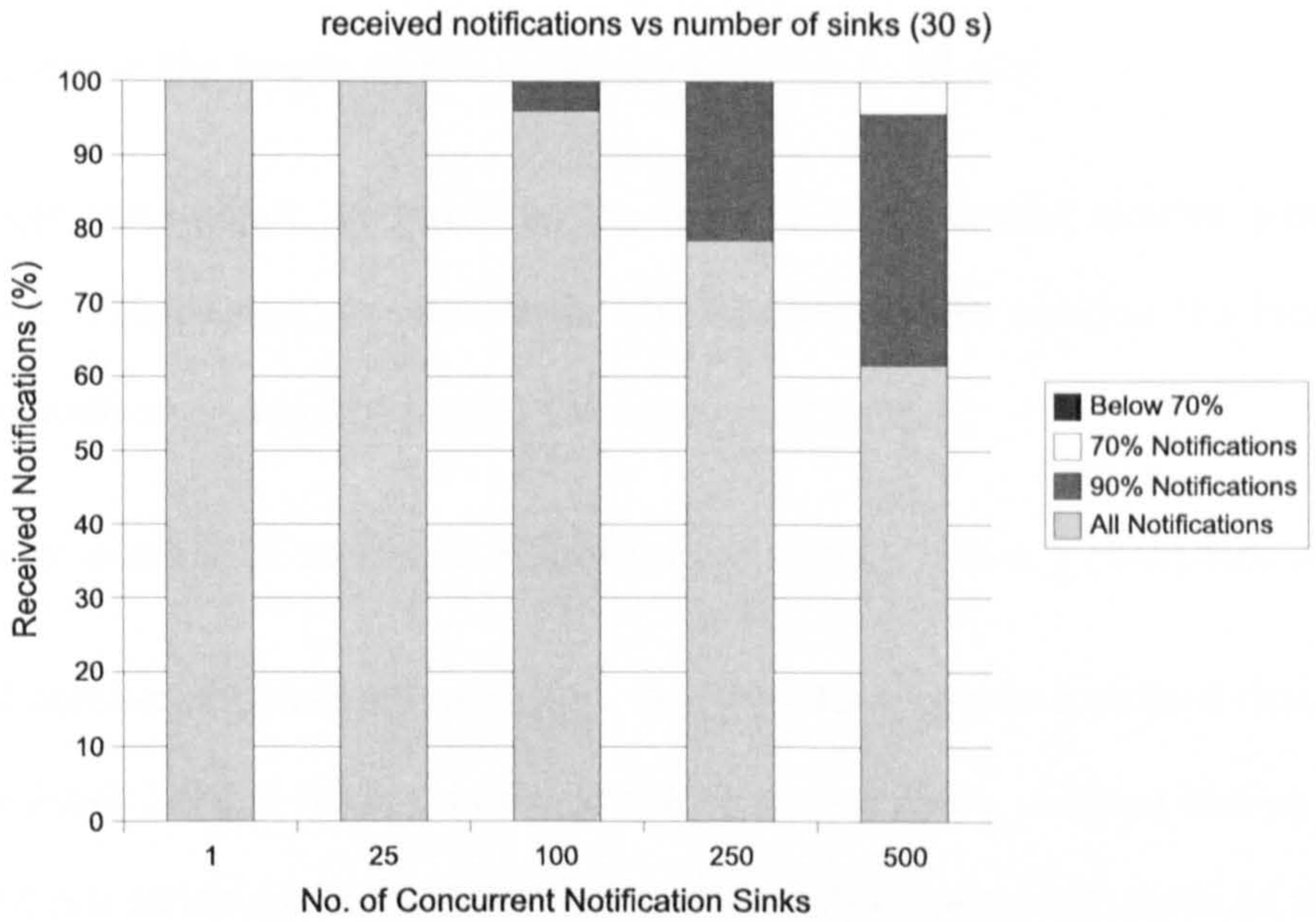

Figure 5.7: Experiment: percentage of received notifications (30 s).

provider until this is updated every time period, as specified. This process, known as *notification caching*, allows SDEs from multiple sources to be aggregated into single sink services. Moreover, the query wait time (time elapsed after query results are obtained, and before issuing the next query) is set to 1 s so as to maximise the number of queries during the experiment duration. The second set of experiments fixes the cache TTL value at 60 s whilst varying the length of time elapsed after results are returned for each successful query and before the next query is issued. The experiments are also each repeated for 10 min.

5.3.2 Performance Metrics

The following performance metrics are used to assess the performance of the Index Service in handling pull-based type queries, from the point of view of both the Index Service and agents:

- Average response time in seconds. This is the average time from sending out a

query and receiving the response, across all the successful queries. The theoretical maximum is the length of the experiment, which is 10 min.

- Average throughput, in terms of the number of successful queries answered per second. This is a server-side metric which demonstrates whether the Index Service scales with an increasing number of concurrent queries.

- Average number of successful responses per agent. This is a client-side metric.

- Total number of successful responses. A successful response is defined that occurring when Index Service results return to the particular client, without timing out. This metric is a server-side one, showing the total number of queries serviced throughout the whole experiment duration.

5.3.3 Experimental Results and Evaluation

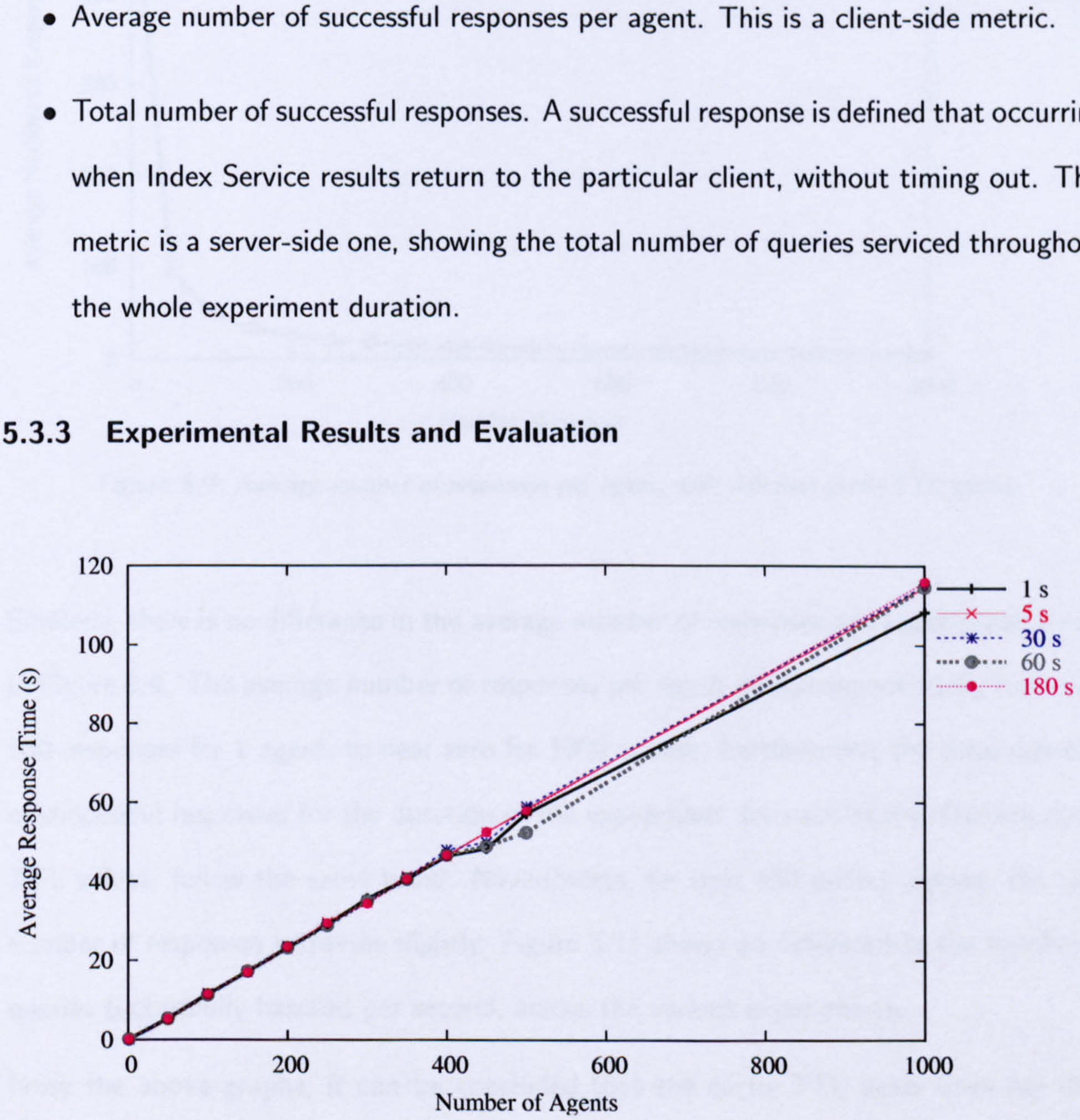

Figure 5.8: Average response times with different cache TTL values.

Figure 5.8 shows the average query response times obtained with the different cache TTL values. The average response time increases linearly for all of the different cache values

until the number of pulling agents exceeds 400. The differences in average response times thereafter is minimal. Therefore, it is observed that the average response time is not influenced by the cache TTL value.

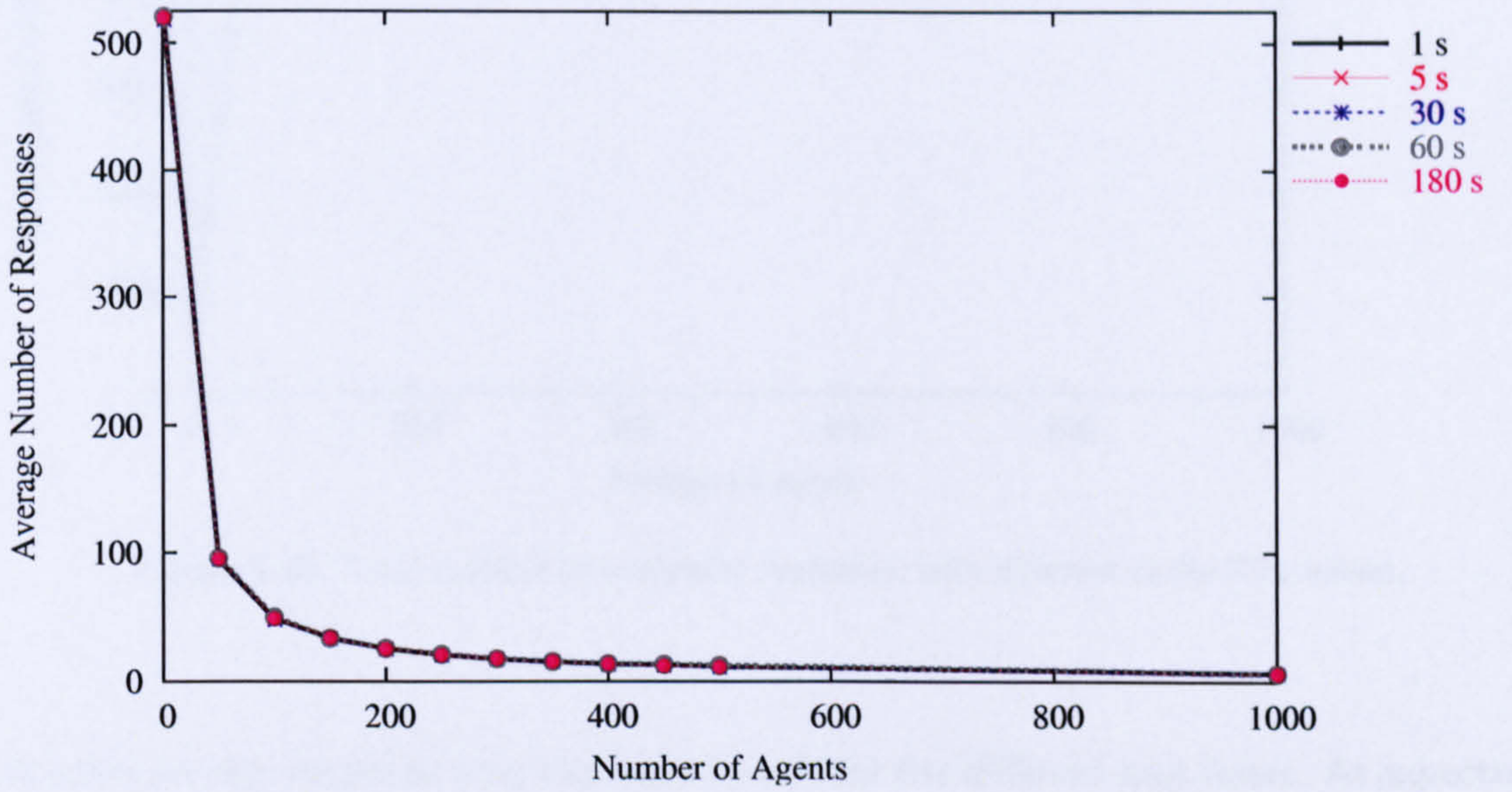

Figure 5.9: Average number of responses per agent, with different cache TTL values.

Similarly, there is no difference in the average number of responses per agent graph shown in Figure 5.9. The average number of responses per agent decays exponentially from over 500 responses for 1 agent, to near zero for 1000 agents. Furthermore, the total numbers of successful responses for the duration of the experiment, for each of the different cache TTL values, follow the same trend. Nevertheless, for over 450 pulling agents, the total number of responses increases slightly. Figure 5.11 shows no difference in the number of queries successfully handled per second, across the various experiments.

From the above graphs, it can be concluded that the cache TTL value does not alter the results obtained from the query process, and therefore a cache TTL value of 60 s is deemed reasonable for the following set of experiments.

Figures 5.12 to 5.14 show the experimental results obtained when the wait time is varied between the receipt of query results and the issue of the next query. Figure 5.12 shows

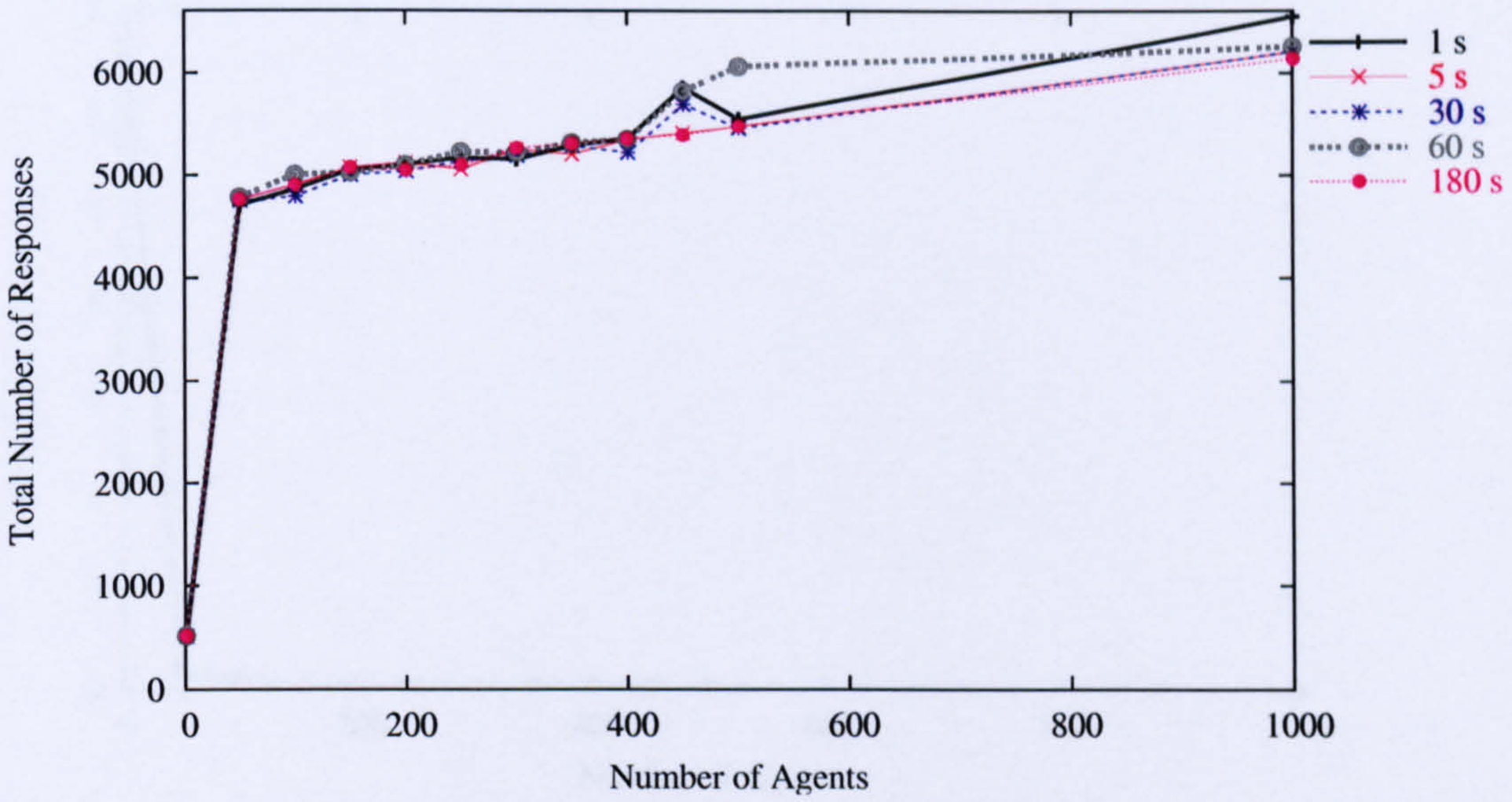

Figure 5.10: Total number of successful responses, with different cache TTL values.

that the average response time increases linearly for the different wait times. As expected, the smallest wait time produces the highest average response time, with 1000 pull-based agents having an average response time of around 115 s. A wait time of 30 s maintains a very low average response time until the number of pull-based agents exceeds around 250 agents. At 1000 pulling agents, the average response time is around 70 s.

The graph in Figure 5.13 show that the average number of responses per agent is rather small as the number of pulling agents increases. It can also be observed that for more than 200 pull-based agents, the average number of responses become more uniform, irrespective of the wait time. For example, for the 30 s wait time graph, 1 agent receives approximately 20 successful responses. However, as more pulling agents compete for the Index Service resources, the average responses decreases, for instance, around 6 responses for 1000 competing agents.

Figure 5.14 depicts the total number of responses for the experiment duration, with different wait times. There is not much difference between the 1 s and 5 s graphs, and for more than 300 pulling agents, the total number of responses is close to that of the

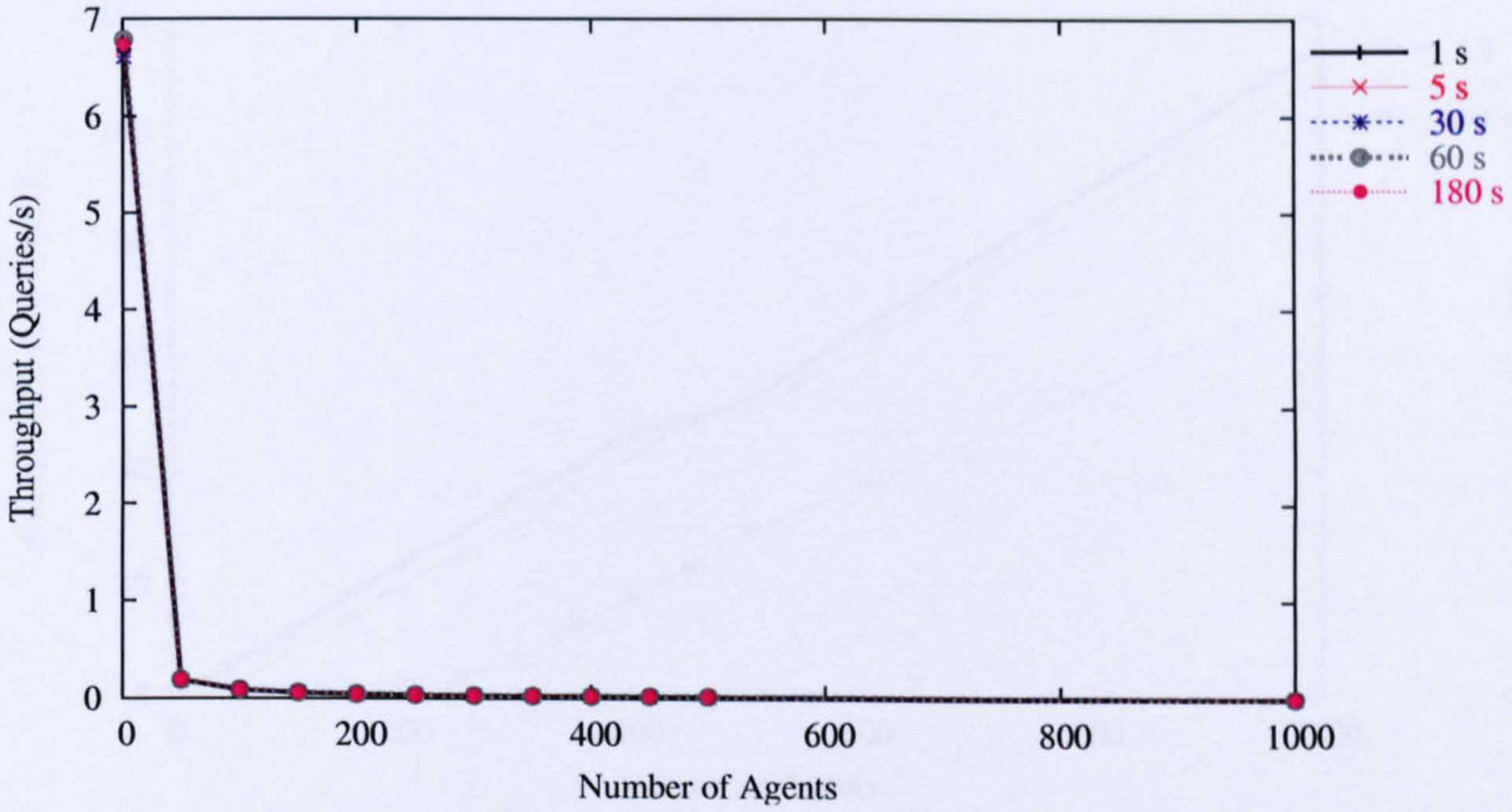

Figure 5.11: Index Service throughput, with different cache TTL values.

other wait time, as the agents compete, the smaller the wait time.

5.3.4 Pull-based Benchmarking Summary

The set of experiments in this section consists of a wait time of 1 s between receiving pull-based query results and issuing the following query. Such a short period is chosen for the wait time so as to stress-test the Index Service as the number of agents also rises. It can be observed that the average response time is the same, irrespective of the cache TTL value, showing that an increasing number of synchronous queries have little effect on the Index Service query handling capability. Accordingly, there is no difference in the average number of responses per agent, across the experiments involving different cache TTL values. However, as the number of agents increases, there is a large drop in the number of successful responses seen by each agent as the agents compete for the Index Service. Moreover, the experimental results show that the Index Service can handle a large number of queries successfully even when the wait time is relatively large, e.g 180 s. Consequently, these experimental results show that a relatively large number of agents

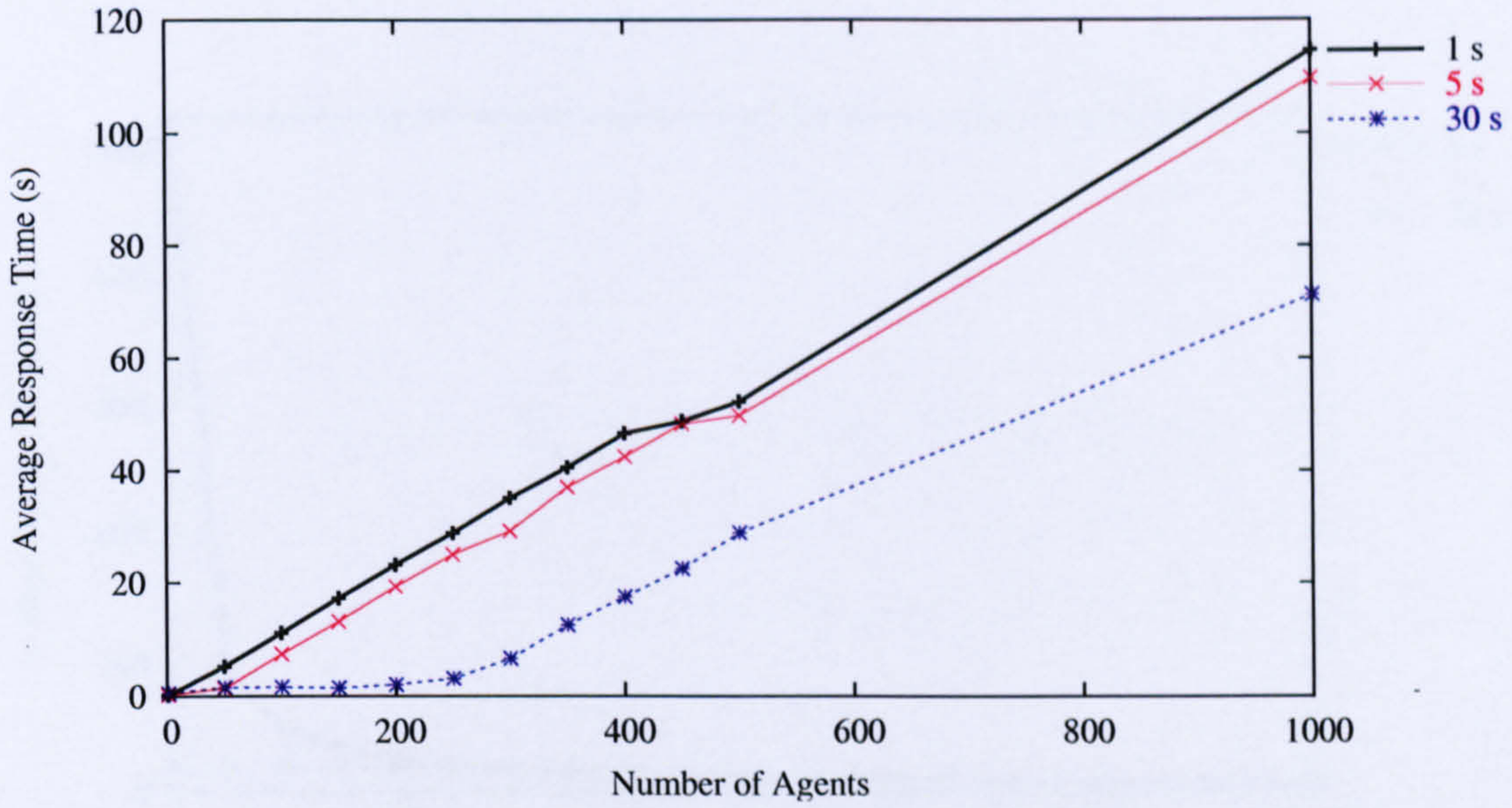

Figure 5.12: Comparison of average response times, with different times after each query and with a fixed cache TTL value of 60 s.

(1000) can query the Index Service repeatedly with a minimum wait time of 1 s, without affecting the server-side performance.

When similar experiments are carried out but with a fixed cache TTL value of 60 s and different wait times, it can be seen that the average response time increases significantly as the wait time decreases. Therefore, if a particular average response time is to be sustained, the wait time between consecutive queries is to be increased. Similarly, for a stable average number of responses received per agent, the wait time should be around 30 s. Additionally, as the number of agents in the experiment increases, the Index Service can service more queries with a shorter wait time, however, this is detrimental to the average query response time. Subsequently, a balance should be struck between the query wait time and the total number of queries serviced.

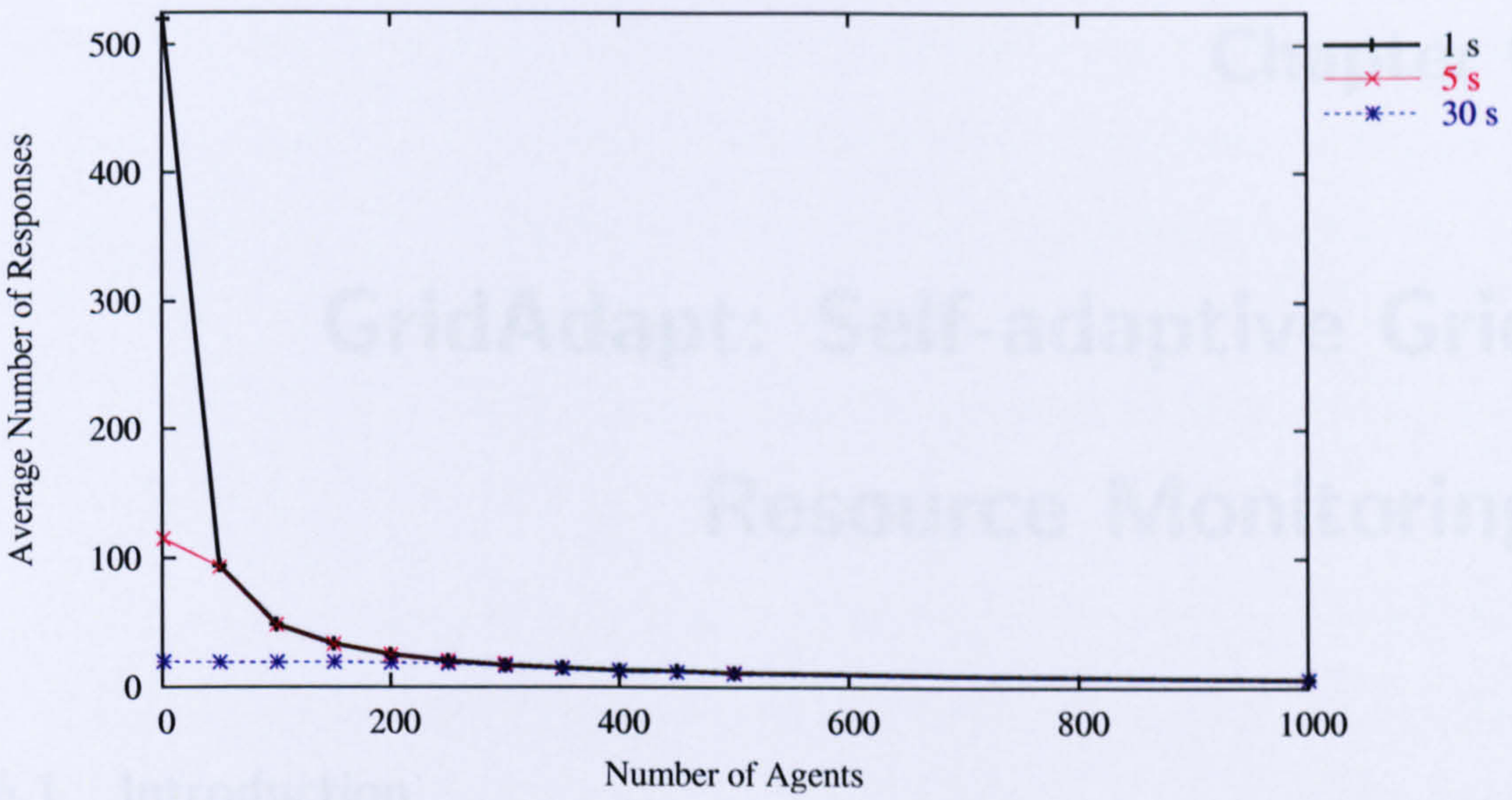

Figure 5.13: Comparison of average number of responses, with different times after each query and with a fixed cache TTL value of 60 s.

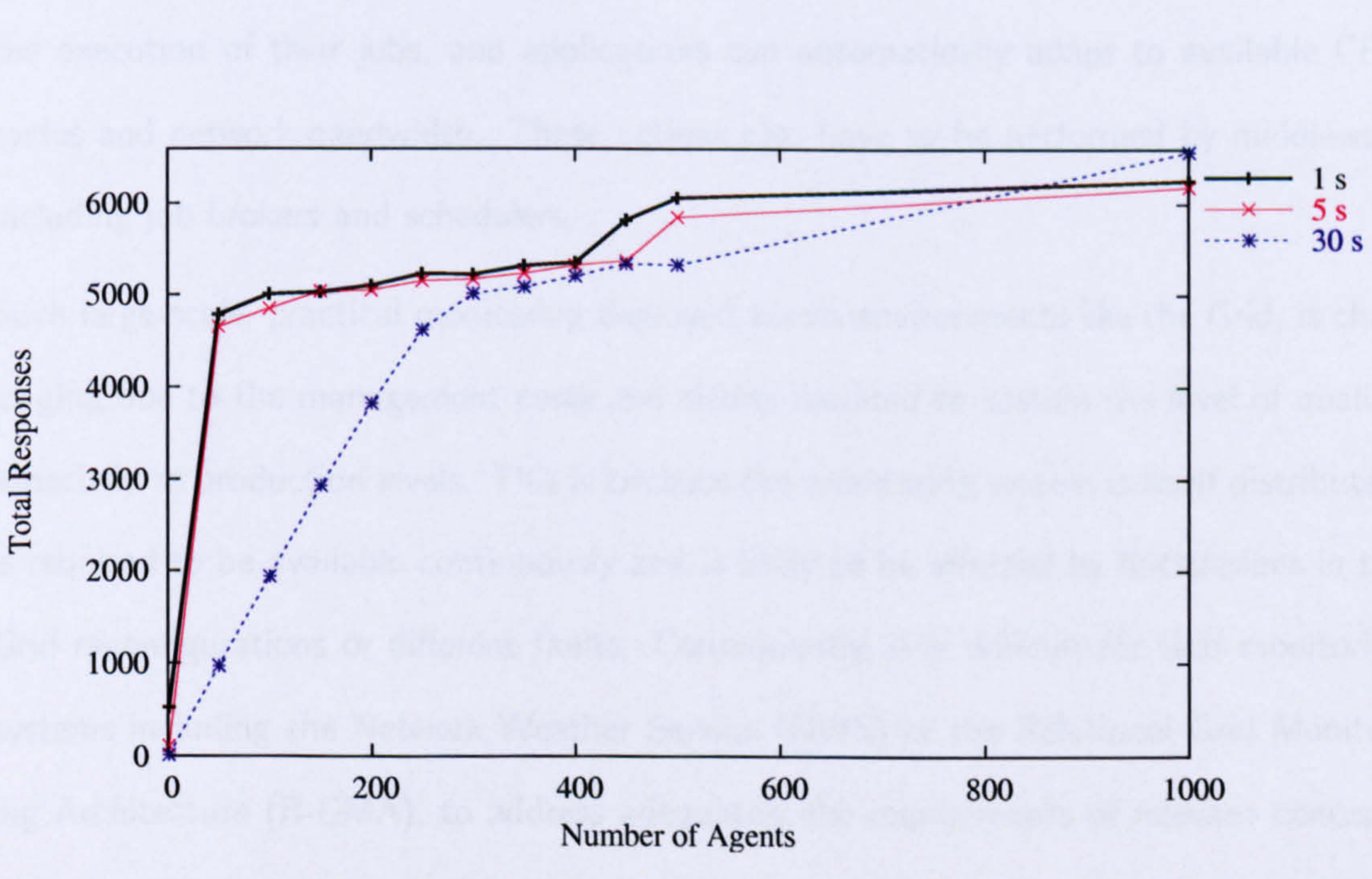

Figure 5.14: Comparison of total number of responses, with different times after each query and with a fixed cache TTL value of 60 s.

Chapter 6

GridAdapt: Self-adaptive Grid Resource Monitoring

6.1 Introduction

Grid monitoring systems are necessary in large deployments of shared resources at all levels. For example, administrators can monitor the status of the Grid, users can plan the execution of their jobs, and applications can automatically adapt to available CPU cycles and network bandwidth. These actions also have to be performed by middleware including job brokers and schedulers.

Such large-scale, practical monitoring deployed across environments like the Grid, is challenging due to the management costs and efforts required to sustain the level of quality, especially at production levels. This is because the monitoring system is itself distributed, is required to be available continuously and is likely to be affected by fluctuations in the Grid reconfigurations or different faults. Consequently, it is difficult for Grid monitoring systems including the Network Weather Service (NWS) or the Relational Grid Monitoring Architecture (R-GMA), to address adequately the requirements of relevant concepts including IBM's *Autonomic Computing*.

The automated management of Grid monitoring systems is inherently complicated by different technical aspects. Firstly, as mentioned above, the components of the monitoring

system are distributed, typically across different administrative domains, and can therefore be prone to faults and reconfigurations. Moreover, Grid monitoring systems typically consist of functional components which are interdependent, and connected by physical networks. As such, system faults and reconfigurations may propagate throughout the whole system, and system modifications are not likely to be effected in isolation. Furthermore, single points of failure are not desirable due to the distributed nature of the monitoring system. Additionally, the performance intrusiveness added to the Grid system by monitoring, should be kept to a minimum.

This chapter builds on the benchmarking results of the MDS3 from the previous chapter, to carry out further performance studies for concurrent push and pull queries. These results are then used to develop several self-adaptive algorithms which form the basis of GridAdapt. Experimental results are also given to show the performance benefits for both the Index Service and the clients.

6.1.1 Autonomic Systems

According to IBM's definition, an *autonomic* system possesses the following eight defining characteristics [49]:

1. It is important for an autonomic computing system to be aware of itself and to have a system identity. Such a system thus needs to know its current status, components and interfaces to other systems it communicates with. In the case of GridAdapt, the contribution work in this thesis, it needs to have an up-to-date aggregation map of resources it is currently monitoring. This feature is important in dynamic virtual organisations [37] where a large number of resources are being shared. GridAdapt is also designed in such a way that it can contribute its knowledge and information to other instances of GridAdapt, on a peer-to-peer basis.

2. An autonomic system must also be able to continually configure its execution when

faced with not only changeable conditions but also future dynamic factors. For instance, an autonomic Grid information service should be able to change its configuration dynamically when faced with an increasing number of clients, each of which has specific QoS demands.

3. Continuous optimisation is also one of the main objectives for an autonomic system, where its various constituent elements are monitored and its workflow consequently readjusted.

4. In addition, fault recovery is one of the other aspects of an autonomic system; the latter should be able to carry on with its operation, following predictable or unexpected events. For example, if a Grid information service becomes severed from a resource which was being monitored, the association must be promptly re-established. Moreover, when an autonomic system anticipates possible issues that may arise, it can readjust the behaviour of its components to ensure normal functioning.

5. Since an autonomic system is open to communication with external entities, it must shield itself against any infiltration that may interfere with its security. This is particularly important when systems, like GIS, share and distribute information about remote resources.

6. For an autonomic system to be able to adapt to changes in its environment, it needs to take in the context of its function and purposefully collaborate with its peers. Subsequently, the nature of its interactions will influence the utilisation of other resources, as well as its own components. Thus, both the autonomic system and its environment change as a result of these cooperations.

7. As an autonomic system has to interact with others in a heterogeneous environment, it must build upon open standards. In addition, this feature should co-exist with its capability to manage itself independently.

8. Another aim for an autonomic system is to manage its resources optimally, as well as components in a user-transparent way. Thus, there should be minimal interaction between the user and the autonomous system.

6.1.2 GridAdapt: Autonomous Configuration of a Grid Monitoring System

The system being proposed in this thesis, provides a general framework for autonomously configuring and managing a Grid monitoring system. Based on information including its current performance, the monitoring system automatically self-configures to deliver up-to-date, dynamic information about resources and network topologies, without affecting the quality-of-service of any of its clients. In designing the framework, attention was paid to the following characteristics:

- *Integration* It should be possible to integrate other existing sensors and Grid monitoring systems with the proposed framework, and obtain resource information from them. This feature will increase the status information collected from distributed heterogeneous resources across multiple administrative domains. Integration is achieved by utilising open-standard middleware, for instance Globus, which supports interoperability across geographically dispersed resources.

- *Scalability* The number of nodes interconnected with complex network topologies, should be scalable. The level of service from the Grid monitoring system should not decrease at the same time. Moreover, the monitoring system should be able to operate ubiquitously with the resources being monitored. It must be possible to add resources without the load increasing uncontrollably.

- *Non-intrusiveness* The process of monitoring should incur low per-node overheads for the resources being monitored. It should be possible for performance monitors to be tuned in terms of CPU, communication, memory and storage requirements.

This is a requirement since high-performance Grid applications have considerable resource demands.

- *Autonomy* The Grid monitoring system should be managed appropriately, for example, autonomously reconfiguring its behaviour to maintain its performance. It should also cater for the dynamic availability of resources, with no user intervention. To do so, the monitoring system should not be rendered inoperable or inaccessible by the resource fluctuations which it is intending to capture.

- *Extensibility* The framework should be flexible enough to allow the addition and integration of different self-management features. The autonomic nature of Grid monitoring systems covers several areas. Firstly, the configuration of the monitoring system should be autonomous, for example, for identifying component dependencies, registering sensors with the directory service, and initialising the sensor, source and sink processes. Although it is not required that these actions occur all at once, the monitoring system should allow for timely reconfigurations to take place as resources join and leave the virtual organisation. Another feature which Grid monitoring systems should have is the prompt detection and handling of faults including the termination of a monitoring process, the loss of node being monitored, or network loss.

- *Portability* The Grid monitoring system should be portable to different operating systems and CPU architectures due to the heterogeneity of resources. The proposed system is based on the Globus Toolkit which acts as middleware for heterogeneous resources. During the early stages of the development of Globus, more emphasis was placed on the Linux platform but later on, other platforms were also considered, including Windows, AIX and HP-UX.

GridAdapt is thus a resource monitoring and selecting system, based on the Open Grid Services Architecture (OGSA). It also employs self-adaptive and self-optimising techniques

to offer a cost-effective, up-to-date client-customised view of the status and availability of loosely-coupled, distributed Grid resources that are often in different administrative domains. In brief, GridAdapt allows the status of heterogeneous, distributed resources to be managed in an automated fashion. Furthermore, the open protocols on which GridAdapt builds, enable it to be autonomous whereby self-regulation and self-management occur with minimal human intervention.

GridAdapt, if applied to a wider scale, allows appropriate MDS servers to be located and selected on the basis of their past performance, relevance and suitability when mapped to clients' requirements. GridAdapt provides answers to questions of the type "Which of two seemingly similar MDS servers can provide me with the best performance?" Given several Grid Information Services, GridAdapt can help clients make informed decisions about the one to register with or query synchronously. In brief, the operation of GridAdapt is based on the collection of performance benchmarks, the use of these benchmarks in its current resource monitoring process, and the self-adaptation and behavioural optimisation in the delivery of its query services.

6.1.3 Overview of GridAdapt

Below are the characteristics that best describe the services which GridAdapt offers, with the aim of transparently providing access to resource information when and where needed. The fundamental characteristics exposed by GridAdapt, making it autonomic, are flexibility, accessibility and transparency [49]. Firstly, resource information should be collected and made accessible in a platform-portable way, as well as shared in dynamic collaborations. Resource information should be up-to-date and accurate. Furthermore, due to the need for omnipresence for a Grid information service, GridAdapt must be accessible all the time. Lastly, transparent GridAdapt operates without any user intervention, as it self-configures and self-optimises to meet its own QoS requirements as well as users'

changeable needs.

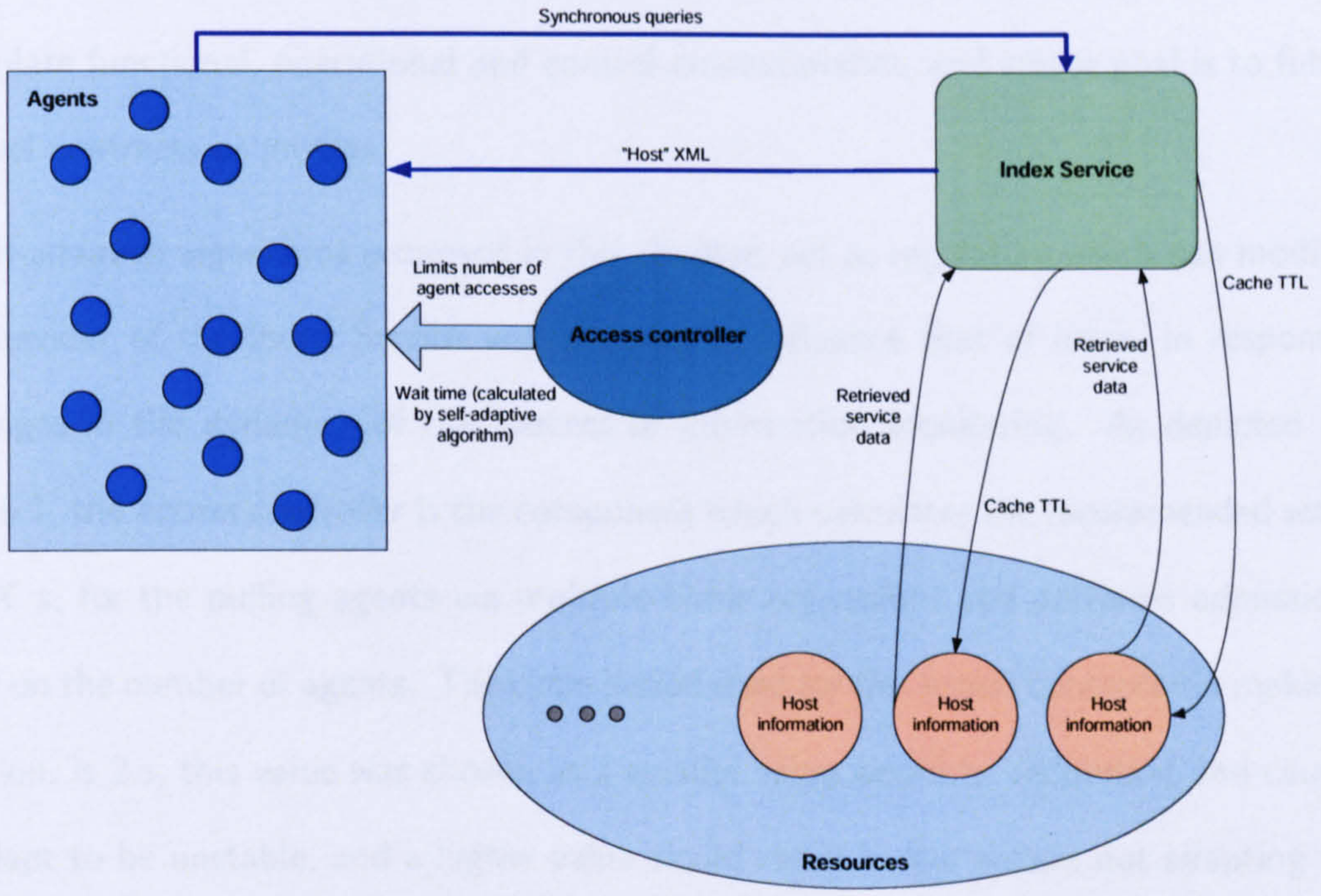

Figure 6.1: Overview of the GridAdapt system, showing pull-based agents.

A conceptual view of the whole GridAdapt system is shown in Figure 6.1. Being built upon the OGSA-based MDS3 (Monitoring and Discovery System), GridAdapt has a number of autonomic components which are defined in terms of programming abstraction interfaces. Information about the behaviour, performance and adaptability of these components is more easily externalised via the interfaces. Such information is used in the dynamic composition and execution of GridAdapt components, based on user policies and server-side constraints. Dynamic composition also takes into account the current residual capacity and workload of GridAdapt. Subsequently, it is crucial for the components to be able to incorporate their environmental conditions into their execution.

In short, GridAdapt provides interfaces to export information about the resources' behaviour, client requirements, and the performance, interactivity and adaptability of Gri-

dAdapt itself. Additionally, system components including sensors, actuators and a decision engine, altogether manage autonomous components to configure, manage, adapt and optimise their execution continuously. In summary, the components making up GridAdapt, encapsulate functional, operational and control characteristics, and whose goal is to fulfil user-level contracts or profiles.

The self-adaptive algorithms proposed in this chapter, act as regulators which can modify the behaviour of the Index Service and attempt to influence that of users, in response to changes in the dynamics of the process of information monitoring. As depicted in Figure 6.1, the *access controller* is the component which calculates the recommended wait time, X s, for the pulling agents via multiple linear regression, and performs admission control on the number of agents. The cycle period used for the access controller in making a decision, is 2 s; this value was chosen as a smaller value would be redundant and cause GridAdapt to be unstable, and a higher value would result in the system not adapting to current conditions fast enough. The wait time value indicates that each agent is advised to sleep for X s after receiving the previous set of query results, before issuing the next query, due to the current residual and load conditions at the Index Service. After X s, the agent queries the Index Service synchronously and subsequently receives the query results in the form of the `Host` XML. Additionally, the cache TTL value is set in the Index Service configuration file and is used to retrieve the `Host` service data element.

6.2 Self-adaptation and Self-optimisation for Push-based Resource Monitoring

This section of the chapter, supported by [70, 71], concentrates on the scalability, reliability and performance of Grid discovery and monitoring services, using autonomous concepts. Part of the research work in this chapter involves application scenarios being developed where the notification rate of data is dynamically modified, based on the overhead costs

at the MDS3 Index Service. Previously collected performance benchmarks, as described in Chapter 5, are also utilised to implement two self-adaptive algorithms which are the basis of the scenarios. The objective of these feedback algorithms is to sustain an adequate level of service for clients while minimising costs at the MDS. Different types of workload models are also used to assess the efficiency of the algorithms. Experimental results are subsequently shown when varying notification update mechanisms and decision parameters are used, ensuring that the MDS is scalable with accruing concurrent clients. Therefore, a policy is proposed where the notification rate is computed dynamically, thereby creating an autonomous Grid monitoring service.

Drawing on the popular Grid Information Service, which is the OGSA-based (Open Grid Services Architecture) MDS3, this chapter proposes a novel approach to prevent the GIS from overloading and to improve its service performance; this is achieved via the dynamic adjustment of the notification rate. This process also leverages the characterisation and evaluation of the performance achievable by the Globus Monitoring and Discovery System (MDS3) which is a widely deployed reference implementation of a Grid information service. The focus of this section of the chapter is therefore the autonomic delivery of dynamic, up-to-date events to clients using the MDS3 push-based mechanism.

6.2.1 Self-adaptive Notification Algorithms

Previous experimental results showed that there is a definite trade-off between providing high data accuracy for the clients and maintaining minimal overhead at the Index Service. Agents acting as brokers to human clients, typically register themselves to the Index Service for the purpose of monitoring resources. To prevent the Index Service from overloading, policies are proposed where the notification rate is dynamically computed, based on the availability of the Index Service and the data accuracy requirements of the agents, thereby creating an autonomous system. These policies are verified using various workloads in the scenario environment. Moreover, the MDS3 performance benchmarks showed that the

notifications which the Index Service sent, quickly reached a saturation point for over 25 concurrent agents, corresponding to a notification rate of 1 s. This saturation value clearly depends on the notification rate. The load overhead experienced by the Index Service is also directly proportional to the number of notification sinks. As this number increases, the self-adaptive notification algorithm dynamically calculates the average upper bound for the notification rate and adjusts the notification rate accordingly. Consequently, both the number of notification sinks and the Index Service availability level influence the overall efficiency of the Index Service.

The aim is for the CPU utilisation to vary between 0.65 and 0.85, which is deemed an appropriate range for the Index Service host to be utilised efficiently without it being overloaded. The load average should be kept at a minimum, with 3.0 being a usual value, as a high load average and a low CPU utilisation can cause problems on the Index Service machine. A high CPU utilisation can also be indicative of excessive paging activity, which can cause a drop in the performance of the Index Service. It is thus useful to use the load average in combination with the CPU utilisation, and that load average be kept to a low value.

Therefore, two different self-adaptive algorithms are proposed: a *sink-based* algorithm and a *utilisation-based* algorithm. The following notation is used in the description of the self-adaptive algorithms:

- *Notification rate* (NR): Represents the rate at which the service data provider is being refreshed. It is also the rate at which notification sinks are being notified.
- The performance metrics which characterise the Index Service overhead most appropriately are the combination of the *load average* and *CPU utilisation*.

Using these metrics, the maximum capacity of the Index Service can therefore be specified using these parameters:

$< LA >$

$< Util_{max} >$

The availability of the Index Service at time *t*, depends on the following model:

$< Util_{curr}(t) >$

The utilisation-based self-adaptive algorithm will make use of the fraction of the Index Service's resources which is occupied to enforce the right decision. The occupancy of the Index Service is thus defined in terms of both the load average and the CPU utilisation which has a maximum value of 100%.

$CU = Util_{curr}(t)/Util_{max}$

- *Number of notification sinks* ($Sinks$): The number of notification sinks currently registered with the Index Service.

The sink-based self-adaptive notification algorithm uses the current number of notification sinks to make a decision on the notification rate. It also uses previously compiled offline performance benchmarks to deduce the optimum notification rate, given the current number of notification sinks. For each condition, the notification rate is calculated based on the optimal rate which will prevent CPU overload.

The utilisation-based self-adaptive notification algorithm bases its decision to change the current notification rate on both the current number of notification sinks and an average value of CU. This average value is calculated dynamically using a moving window of the last ten CU readings throughout an application scenario. This moving window size is chosen as it gives an up-to-date representation of the past data values which are neither too far in the past, nor inadequate for prediction. The load average is also taken into consideration and for a value which is too high, the notification rate is correspondingly decreased. Moreover, this algorithm also uses previously compiled offline performance benchmarks to deduce the optimum notification rate change given the current number

of notification sinks. These benchmarks are used to profile the Index Service for the platform under consideration. Furthermore, decisions take into account the comparison of the current CU to its previous value, as well the current load average value. When CU is low, the notification rate is set to the optimal value which is obtained from the previously collected performance benchmarks because the load on the Index Service is low. On the other hand, when CU is high, the notification rate is decreased by an optimal value for reducing the load on the Index Service. Furthermore, when CU is in the *average range*, the notification rate increase depends on the number of sinks. The greater the number of sinks, the smaller the notification rate increase, to reduce the overall load on the Index Service. During each cycle of the algorithm, the load average is checked for being too high.

The CPU utilisation-based self-adaptive notification pseudo-code algorithm is as follows:

```
switch(CU) {
case (average):
   if (Prev = average & Prev > CU  || Prev != average) {
      if (LA > optimal) {
         switch(Sinks) {
            case many   : Set NR to NR + a;
            case few    : Set NR to NR + b;
            case medium : Set NR to NR + c;
         }
      }
   }
case (low):
   if (Prev = low & Prev > CU || Prev != low) {
      if (LA > optimal) {
         switch(Sinks) {
            case many   : Set NR to NR + d;
            case few    : Set NR to NR + d;
            case medium : Set NR to NR + d;
```

```
            }
         }
      }
   case (high):
      if (Prev = high & Prev > CU || Prev != high) {
         if (LA > optimal) {
            switch(Sinks) {
               case many   : Set NR to e;
               case few    : Set NR to minimum;
               case medium : Set NR to f;
            }
         }
      }
   }
```

where

$$Sinks(t) = \begin{cases} many & \text{if } Sinks(t) > 250 \\ few & \text{if } Sinks(t) < 25 \\ medium & \text{if } 25 <= Sinks(t) <= 250 \end{cases}$$

$$CU(t)) = \begin{cases} low & \text{if } CU < 0.25 \\ average & \text{if } 0.25 <= CU <= 0.85 \\ high & \text{if } CU > 0.85 \end{cases}$$

$$NR = \begin{cases} minimum = & 1s \\ maximum = & 150s \end{cases}$$

$$LA = \begin{cases} optimal = & 3.0 \end{cases}$$

Values for the constants a to f are optimal, drawn from the benchmarking data, that would prevent the Index Service from overloading.

6.2.2 Application Workload Scenarios

An application scenario environment has been developed to experimentally verify the benefits and cost-effectiveness of the two proposed self-adaptive notification algorithms. The scenario environment is multi-threaded and it consists of several components including the notification sinks, the Index Service and a decision-making engine. Additionally, the cycle of the self-adaptive algorithm is 10 s; on each execution, the notification rate is either modified or maintained, depending on the overhead at the Index Service.

Experimental Agent Workloads A workload is an agent application which executes as part of the application scenario. Two types of workload models have been set up for the experiments. Firstly, an *increasing* workload helps to gauge the limits of the Index Service capacity. At the beginning of the application scenario, only one notification sink registers with the Index Service. Subsequently, after every 30 s, eighteen additional agents are registered for notification. A second *dynamic* workload was applied, where the number of notification sinks registered to the Index Service, is modified during the course of the application scenario. Therefore, every 30 s, the number of sinks was randomly increased or decreased by a number between 1 and 75. The starting value of the notification rate for both workloads is 1 s.

6.2.3 Experimentation and Results

Experiments were performed to verify and quantify the benefits of using the self-adaptive notification algorithms.

	a	b	c	d	e	f
NR (seconds)	20	5	10	45	50	20

Figure 6.2: NR values for constants a to f.

Experimental Setup

The experiments were carried out on a Globus Toolkit 3 Grid testbed; this was the current latest stable version of the Grid middleware software at the time of writing. Across the experiments, an agent representing a Grid application, acted as a notification sink. In short, during the experiments which each last ten minutes, the performance of the Index Service was monitored as the application scenario ran. Performance data was collected from a series of ten experiments, and the average results are shown in the following section. Conclusions are then drawn from the quantitative advantages of the proposed self-adaptive notification algorithms.

The agents were written using the OGSA 3.0.2 APIs [47] and they subscribed to the Index Service for changes in one of its SDEs. The simulator ran on a different host from the Index Service (same set-up as in the first footnote in Section 4.5.3). In these experiments, the client machines are connected to the Index Service host. Several characteristics were observed on the Index Service, namely its CPU load and utilisation, as the application scenario was carried out. Additionally, the Index Service was configured on a Linux kernel 2.4.18-14 machine with a 1.9 GHz processor and 512 MB RAM, and it was part of a Tomcat container, version 4.1.27.

The values used for constants a to f in the utilisation-based self-adaptive algorithm for the following experiments are given in Figure 6.2.

Performance Metrics

By using an increasing workload and then a dynamic one, the situation whilst the self-adaptive algorithms are used, is compared with the case when no algorithm is used. The following performance metrics are used to assess the performance of the Index Service in sending notification messages to its clients:

- 1 min CPU load average ($\mathcal{L}_1$). The load average is indicative of the Index Service being under heavy usage, which decreases the notification rate and can result in time-outs.

- Percentage of CPU utilisation ($\mathcal{CPU}_{util}$). CPU utilisation is a function of the load on the Index Service, and a high value indicates that the Index Service host can no longer support more notifications and performance will therefore drop.

Experimental Results

Experiments have been conducted to evaluate the two proposed self-adaptive notification algorithms when subjected to the two types of workload models. The first set of experiments runs the application workload without any self-adaptive algorithm and both the second and third sets use the algorithms. Experimental results are shown, detailing the impact on the performance of the Index Service. The performance of the self-adaptive algorithms in terms of the CPU load, is analysed.

- **CPU Load Across the Experiments** Figure 6.3 shows how the CPU load changed with different self-adaptive algorithms and workload models.

The CPU load is measured as the average number of jobs in the run queue. For the increasing workload with no self-adaptive algorithm, it can be seen that as more notification sinks are being added at the rate of eighteen every 30 s, the CPU load generally increases

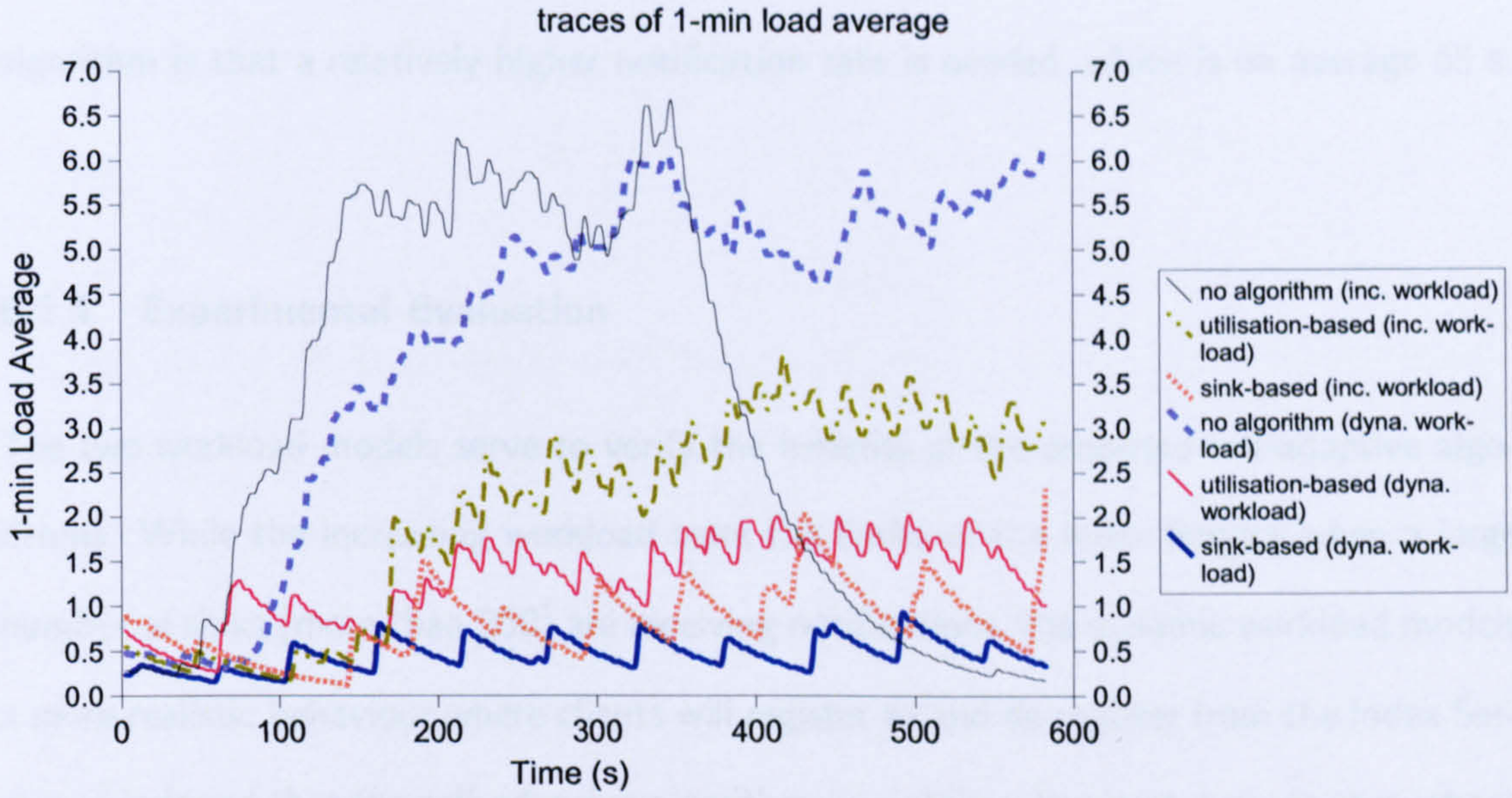

Figure 6.3: CPU load for all experiments.

to reach an average of 5.5.

When the sink-based self-adaptive algorithm is used with the increasing workload, the general trend for the CPU load is a consistent decrease followed by an increase. The periods of increase correspond to the addition of more notification sinks; the most significant increases happen when both increasing numbers of sinks are being registered to the Index Service and when the notification rate is being changed. To counter such load increases, the sink-based self-adaptive algorithm maintains a relatively small load of 0.7. In contrast, for the utilisation-based algorithm with increasing workload, the CPU load is maintained on average at about 2.8.

Furthermore, the CPU load was on average 1.3 and indicated very slight changes for the utilisation-based self-adaptive algorithm with dynamic workload. Overall, it can be observed that the highest loads are produced when no self-adaptive algorithm is used. Additionally, the lowest loads are obtained when the sink-based algorithm is utilised with either workload model. This can be explained by the choice of optimal values for the notification rate which will minimise the CPU load for a given number of notification sinks,

as shown by previously collected benchmarks. However, the drawback of the sink-based algorithm is that a relatively higher notification rate is needed, which is on average 65 s.

6.2.4 Experimental Evaluation

The two workload models serve to verify the benefits of the proposed self-adaptive algorithms. While the increasing workload tests the limits of the Index Service when a large number of sinks (more than 250) are receiving notifications, the dynamic workload models a more realistic behaviour where clients will register to and de-register from the Index Service. t is found that the self-adaptive algorithms maintain a low load average throughout the experiments. It is also found that with the increasing workload, the average notification rate throughout the experiment was 36.2 s and the average CPU utilisation was 37.16%. These results are promising because the accuracy of the service data is relatively high (updated every 30 s) and the CPU utilisation could be increased to near its maximum capacity. On the other hand, the dynamic workload showed an average notification rate of 23.91 s and the CPU utilisation averaged 25.84%. Moreover, the average number of concurrent sinks was around 132, indicating that the Index Service can be adapted to prevent system overload, with the clients having a minimal notification rate. Such experimental results show that it is possible to increase the Index Service utilisation to a predefined value with no drastic decrease in the notification rate. Subsequently, more sinks can be registered when self-adaptation is in place.

The sink-based algorithm maintained a much lower load than the utilisation-based one, indicating that it is a reliable method for controlling the overall utilisation of the Index Service. However, this is not a scalable solution as the Index Service would need to be modelled to discover the optimum values for the algorithm. Nevertheless, the utilisation-based method results in a higher notification rate, which is useful for making sure clients receive the best possible service level. It is therefore believed that a combination of both

algorithm mechanisms works well to ensure that the Index Service provides a reliable level of performance, whilst being self-optimising.

6.2.5 Summary

This section addresses the implementation of a self-adaptive mechanism to maximise the performance of the Index Service from a Grid application's point of view in an autonomous manner, based on previously collected performance data benchmarks. The efficiency of these algorithms is also exhaustively verified, using two different workload types: an increasing and dynamic workloads. The experimental results show that both workload models prove to benefit from the proposed self-adaptive algorithms; the consistency margin of the data being monitored is relatively low and a fairly large number of concurrent sinks can be supported by the Index Service.

6.3 Self-adaptation and Self-optimisation for Both Push- and Pull-based Queries

The Index Service allows a combination of both asynchronous and synchronous queries. Whilst some form of admission control is necessary to prevent the Index Service from exceeding its residual capacity, the question arises concerning the balance between push and pull queries. This section of the thesis enables the optimisation of the query-support mechanism of the Index Service by self-adapting according to the current number of pull-based agents as well as the optimal average query response time which is expected on the client side. Additionally, from the Index Service point of view, the self-adapting algorithm proposed ensures that an adequate number of successful queries are handled on average.

This section proposes a novel self-adaptive algorithm which integrates both push and pull data dissemination by the Index Service, and analyses its performance with a large number of querying clients. Using this algorithm, several push and pull strategies are enabled. A

series of experiments and their results are shown in the following sub-sections, involving a combination of push and pull querying conditions which affect the residual capacity of the Index Service as well as the quality-of-service observed by the clients.

6.3.1 Experimental Setup

For the rest of this chapter, the experiments are carried out on a Globus Toolkit 3 Grid testbed where an agent representing a Grid application, queries the Index Service synchronously. During the experiments which each lasts ten minutes, the performance of the Index Service is monitored as different types of workloads run. Performance data is collected and the average results are shown in the following sections. Conclusions are then drawn from the quantitative advantages of the proposed self-adaptive algorithm and mechanism.

The push and pull agents are written using the OGSA 3.0.2 APIs [47] and they run on a machine with the following specifications: a 667 MHz processor with the Linux operating system installed with kernel 2.4.22-1.2174.nptl and 448 MB RAM. Additionally, the Index Service is configured on a Linux kernel 2.6.12-1.1372_FC3 machine with a 1.9 GHz processor and 512 MB RAM, and it is part of a Tomcat container, version 4.1.27. The admission controller which comprises the sink and pull controllers, runs on a separate machine with these specifications: a 1.5 GHz processor with the Linux operating system installed with kernel 2.6.12-1.1372_FC3 and 256 MB RAM. This is done so that the load on the Index Service machine does not become affected. All the hosts are on an Ethernet LAN and are connected to the Index Service host by a 100 Mb link.

6.3.2 Benchmarks of Push and Pull Queries

To compare the performance of the Index Service when both push- and pull-based queries are issued by clients, similar experiments as in Section 5.2 are carried out. Nevertheless, in

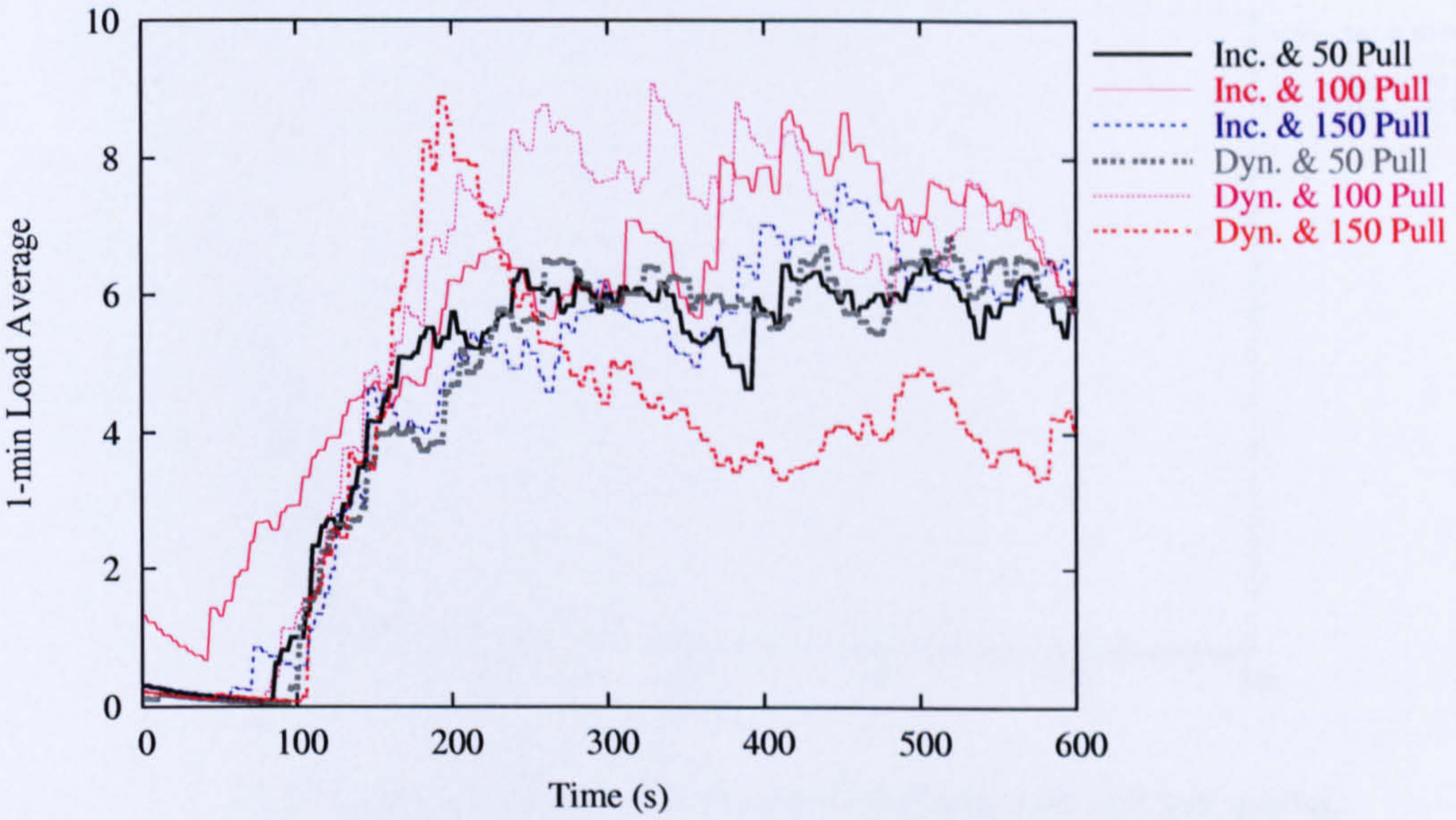

Figure 6.4: 1 min load average for both push and pull queries.

this sub-section, the set of experiments also includes up to 150 agents querying the Index Service synchronously. The latter have a waiting time of 30 s between receiving query results and issuing subsequent queries. The cache TTL of the Index Service is set to 60 s for the rest of this chapter, as the cache TTL does not influence the client-side performance as shown in the previous chapter. Moreover, a number of asynchronous agents are registered with the Index Service and their registration and de-registration patterns follow the *increasing* and *dynamic* workloads which have been described in Section 6.2.2. The aim of these benchmarks is to examine the effect on the Index Service host of typical patterns of query. The experimental results with respect to the Index Service are given below and analysed.

As illustrated in Figure 6.4, for the duration of the 10 min experiment, the 1 min load average was rather high and ranged between 5 and 8. On average, the increasing workloads results in a higher load than the dynamic ones, following similar behaviour as in Section 6.2.3. Furthermore, the 1 min load average increases significantly 100 s into the experiment as a larger number of sinks register themselves to the Index Service. The slight

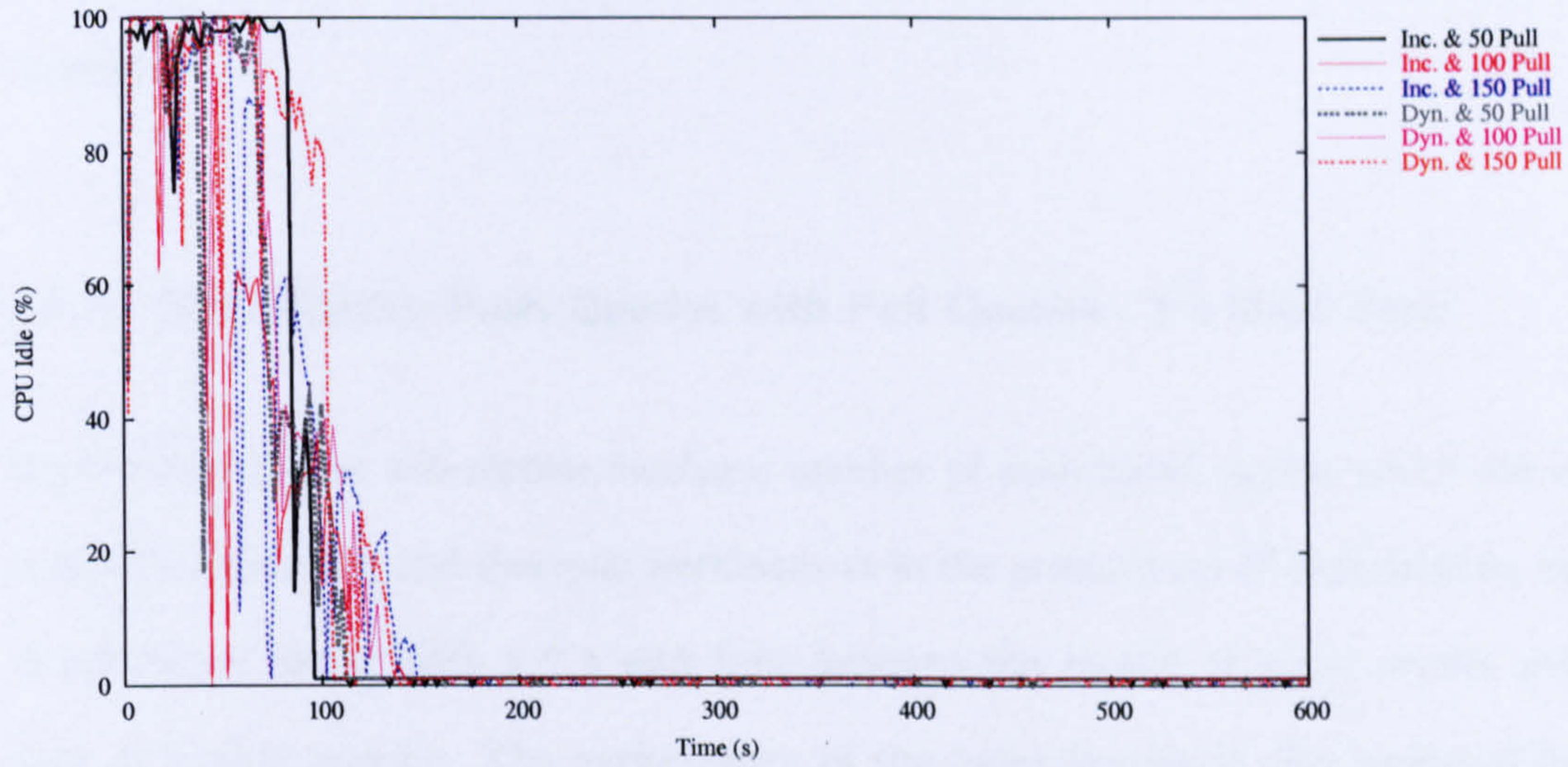

Figure 6.5: Percentage of CPU idleness for both push and pull queries.

decrease in load average towards the start of the experiment can be attributed to the Index Service stabilising itself after the initial surge of queries. Moreover, for each of the two workloads, the load average is higher as the number of pulling agents increases. However, the load average with 150 pulling agents is lower than that with 100 pulling agents, indicating that not all the sinks are succeeding in registering with the Index Service.

Figure 6.5 shows the percentage of the CPU which is idle on the Index Service host. Initially, the CPU idleness remains fairly stable near 100% but after 100 s during the experiment, CPU idleness drops to near 0%. Moreover, for each workload, the larger the number of synchronous agents, the bigger the drop in CPU idleness. Additionally, for the same number of synchronous agents, the drop in the increasing workload is greater than that in the dynamic workload, as overall, more sinks are involved.

The percentage of memory used on the Index Service host throughout the experiment is shown in Figure 6.6. The change in the memory utilisation follows similar trends for the different scenarios, with a steady rise past 100 s into the experiment, followed by a marked decrease in the rise of memory used. However, the memory utilisation is relatively high, levelling off mostly around the 25% mark, with a maximum of 150 pulling agents.

For each of the two workloads, the memory used increases with more pulling agents, as expected.

6.3.3 Self-adaptive Push Queries with Pull Queries - 5 s Wait Time

Experiments in this sub-section involve a number of push-based agents which are represented by *increasing* and *dynamic* workloads as in the previous set of experiments, as well as pull-based agents with a 5 s wait time between the receipt of query results and the issue of further queries. The performance of the Index Service is also improved by applying the self-adaptive notification algorithm in Section 6.2.1 to the push-based agents. The following graphs depict the performance results obtained from a set of repeated experiments.

Figure 6.7 shows the 1 min load average experimental results obtained. It can be observed that the load average increases with the addition of pulling agents; however, for more pull-based agents, the load drops significantly and oscillates around 0.2. The reason behind this behaviour is the inability of the Index Service host to handle more than 50 pull-based agents, each with a 5 s wait time. When there are more than 50 pull-based agents, the Index Service gives priority to the latter, at the detriment of sinks. This behaviour can be seen in Figure 6.29. Similar experiments are carried out, except for a dynamic workload for push-based agents. As observed in Figure 6.8, the 1 min load average follows similar patterns to that with the increasing workload, except that the load is much lower, as expected for a dynamic workload. Frequent peaks in the load average indicates the points at which sinks are being added to the Index Service. On the other hand, the load increases for the addition of pulling agents are negligible.

Figure 6.9 shows a summary of the CPU idleness distribution which can be found in Appendix E. The mean, mode and minimum of the CPU idleness values are shown across the experiments with different numbers of pulling agents. It can be observed that without

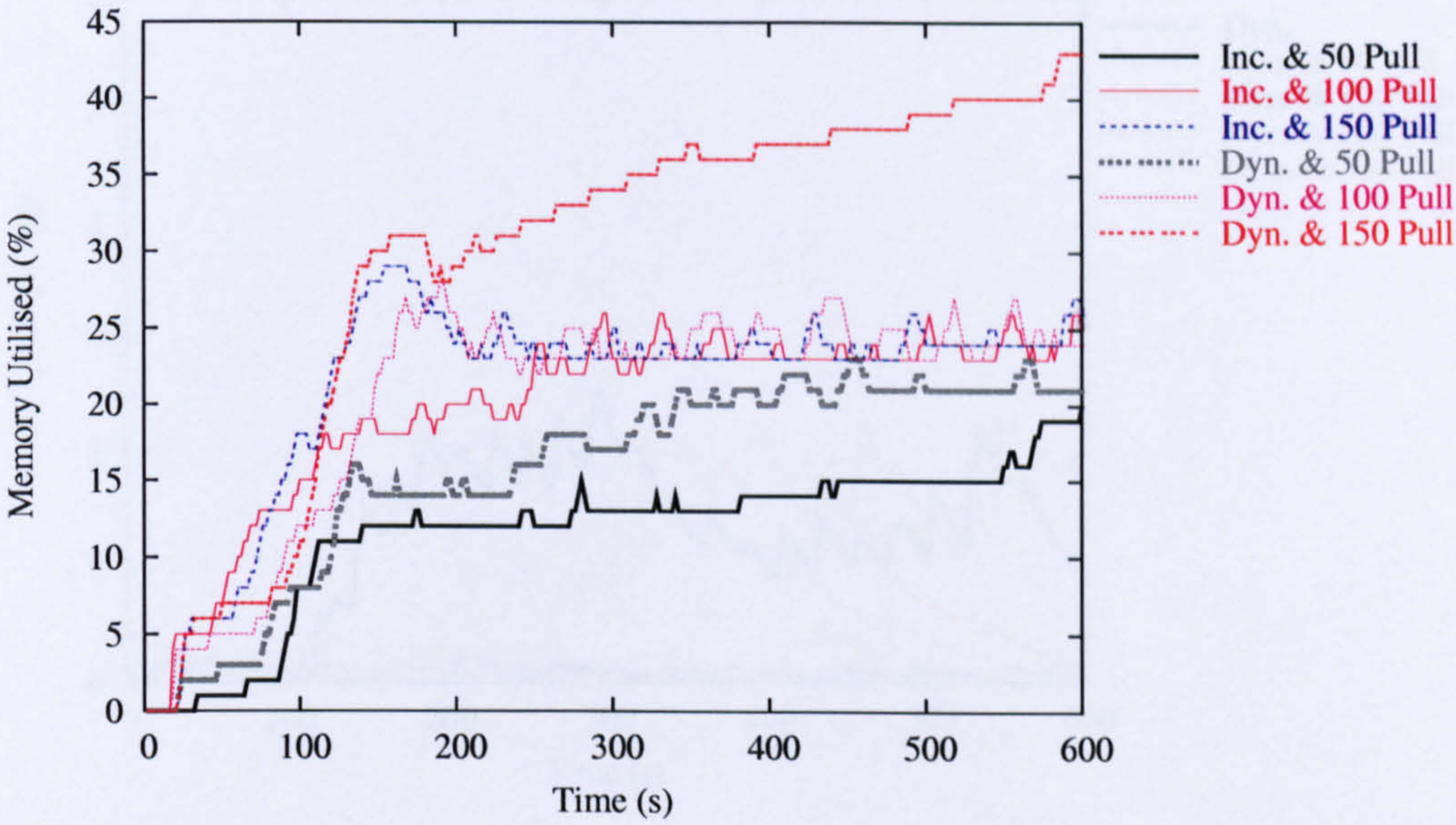

Figure 6.6: Percentage of memory utilised for both push and pull queries.

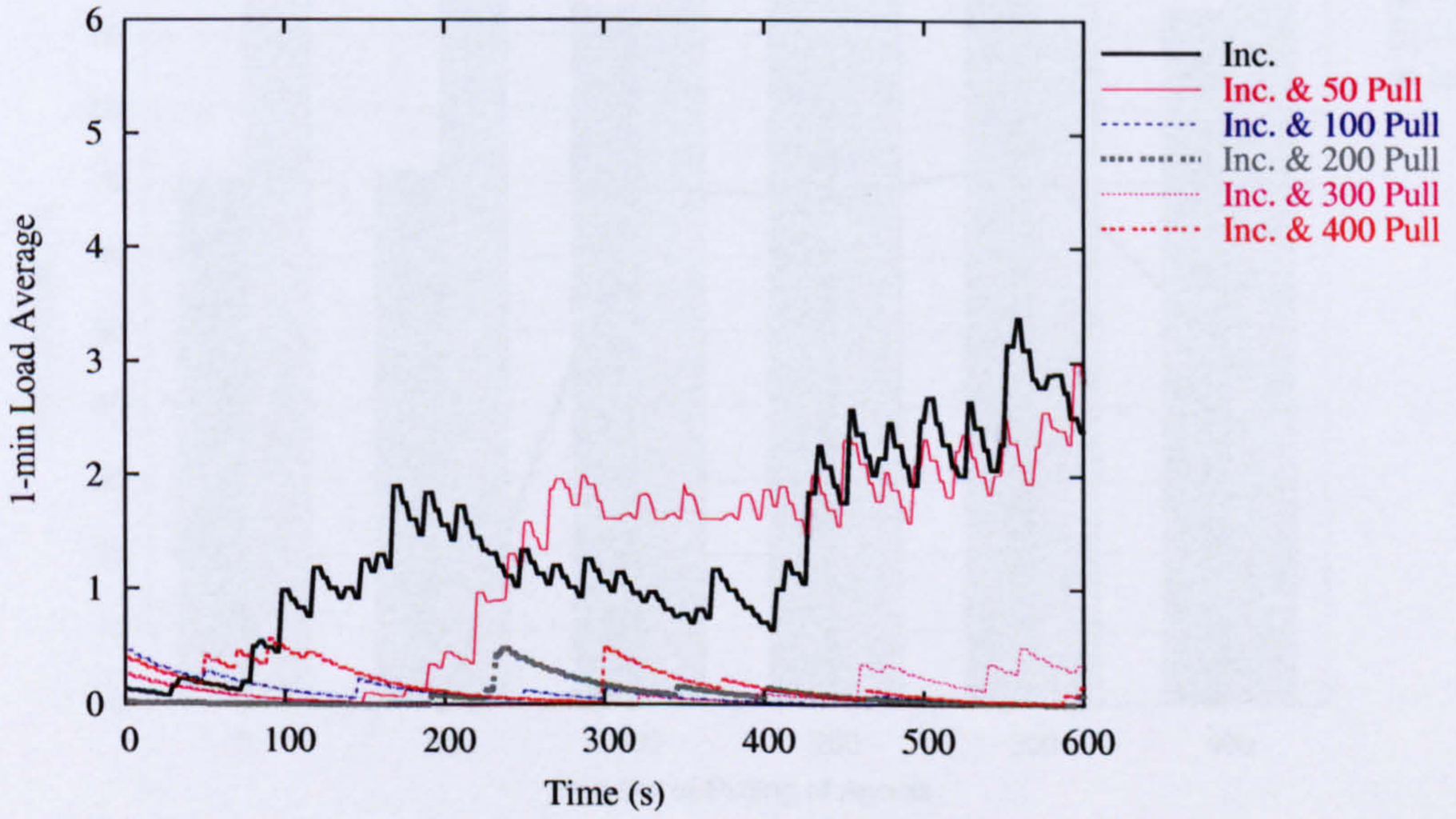

Figure 6.7: 1 min load average for self-adaptive push agents (increasing workload) and pull agents with a 5 s wait time.

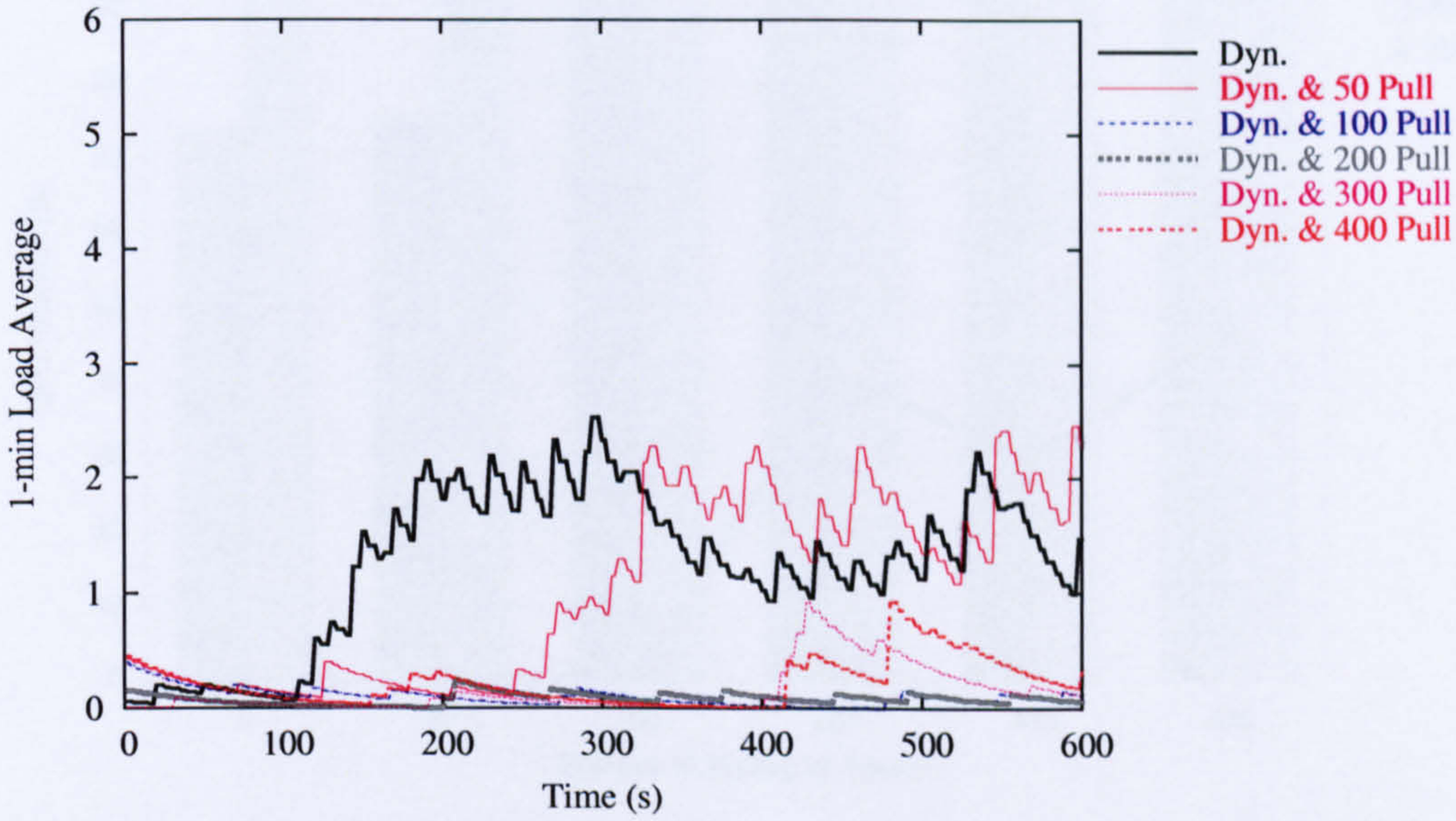

Figure 6.8: 1 min load average for self-adaptive push agents (dynamic workload) and pull agents with a 5 s wait time.

Figure 6.9: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents with a 5 s wait time.

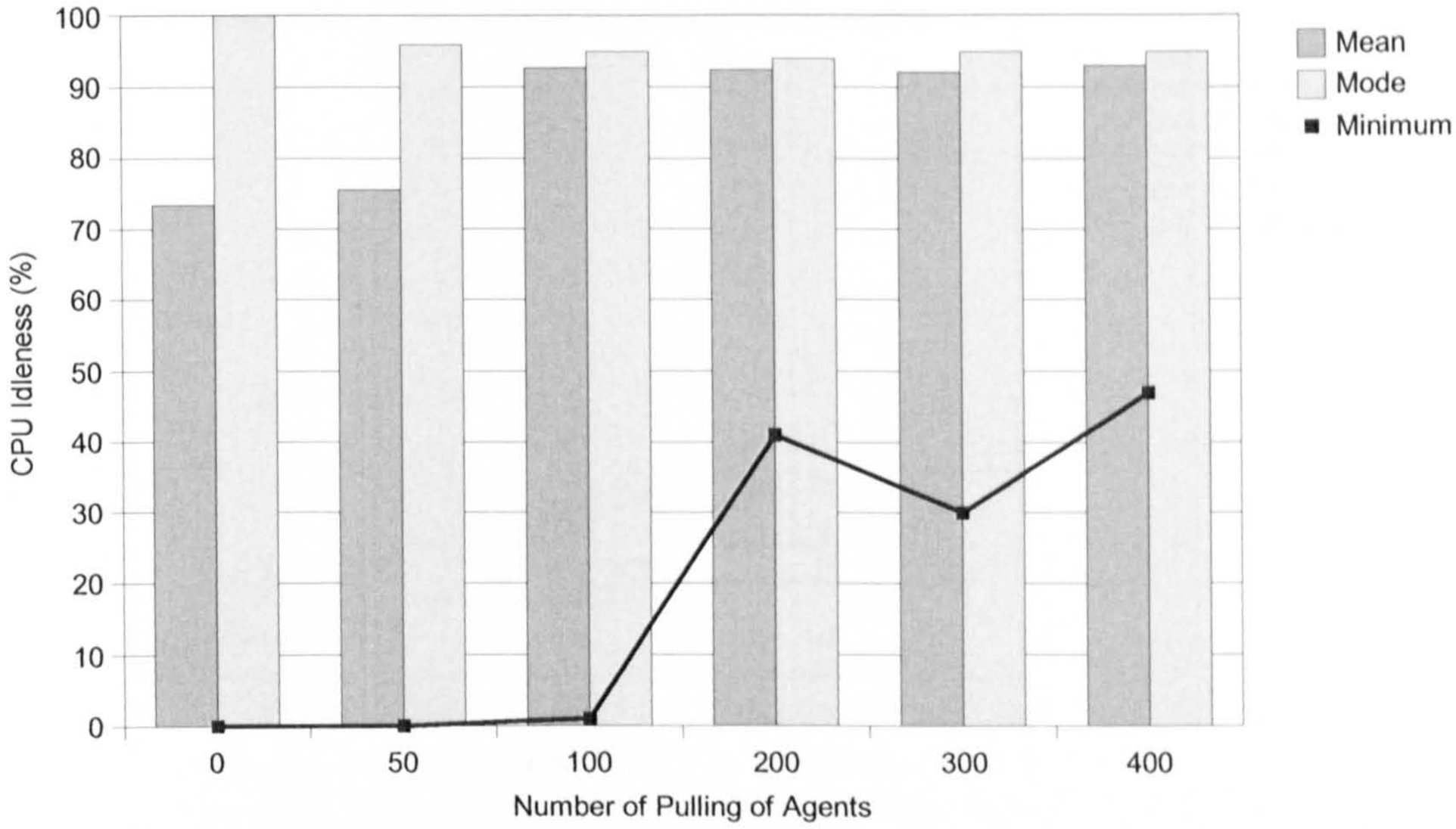

Figure 6.10: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents with a 5 s wait time.

any pull-based agents, the CPU idleness mostly remains near 100%, with dips when sinks are being added, bringing the CPU idleness average down to around 71%. There is also around 5% drop in CPU idleness when pull-based agents are added to the system. Moreover, the drop in CPU idleness generally increases with the number of pull-based agents (as shown by the minimum value), until there are more than 50 such agents. Similar trends are present in the experimental results as shown in Figure 6.10 for a dynamic workload, except for more CPU activity indicated by greater changes in CPU idleness.

As illustrated in Figure 6.11, the usage of memory on the Index Service host increases uniformly with an increasing number of sinks being registered. As expected, more memory is used when pull-based queries are introduced, rather than having solely push-based ones. The difference in memory usage can be up to around 8%. Additionally, memory utilised is the highest with 50 pulling agents and an increasing sink workload, tying in with the 1 min load average graph above. The same linear increases in memory used are found in Figure 6.12, but compared with the increasing workload, the increases are generally

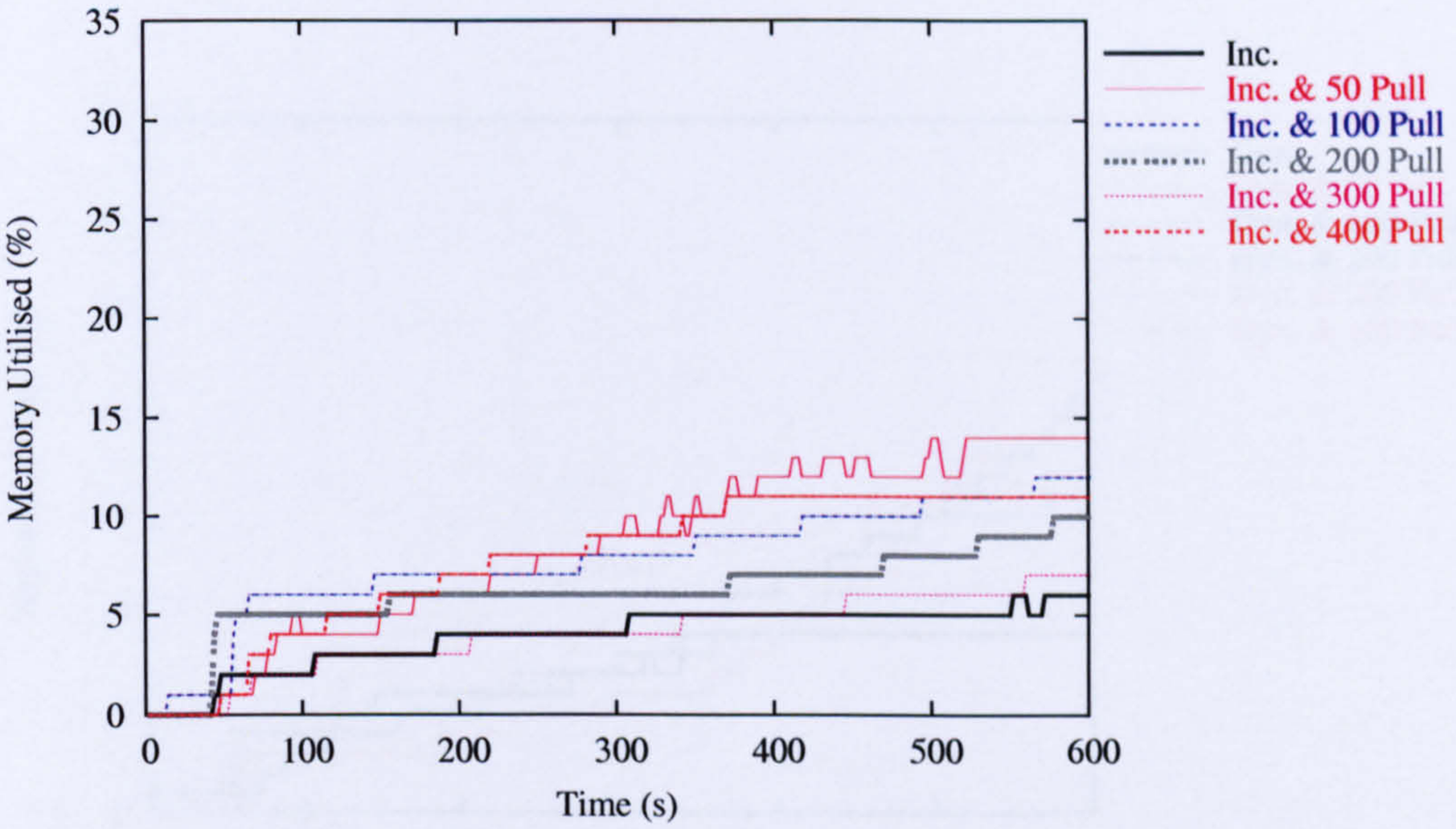

Figure 6.11: Percentage of memory utilised for self-adaptive push agents (increasing workload) and pull agents with a 5 s wait time.

higher.

6.3.4 Self-adaptive Push Queries with Pull Queries - 30 s Wait Time

In this sub-section, similar experiments are carried out as in Section 6.3.3; however, here the wait time is 30 s from the time of receipt of query results and issuing the following one. Figure 6.13 demonstrates that the 1 min load average gradually increases for the duration of the experiment, except when there are 400 pull-based agents. These results show that increasing the wait time to 30 s, does not result in overloading the Index Service, when compared with the 5 s wait time results. The overload does however happen at 400 pull-based agents. Furthermore, all the experiments result in uniform increases in 1 min load average in Figure 6.14, for the dynamic workload. The number of each type of agents serviced during the experiment can be observed in Figure 6.29.

Figure 6.15 summarises the CPU idleness distribution in Appendix E and shows that the CPU is more idle across the length of the experiment, when compared with the 5 s wait

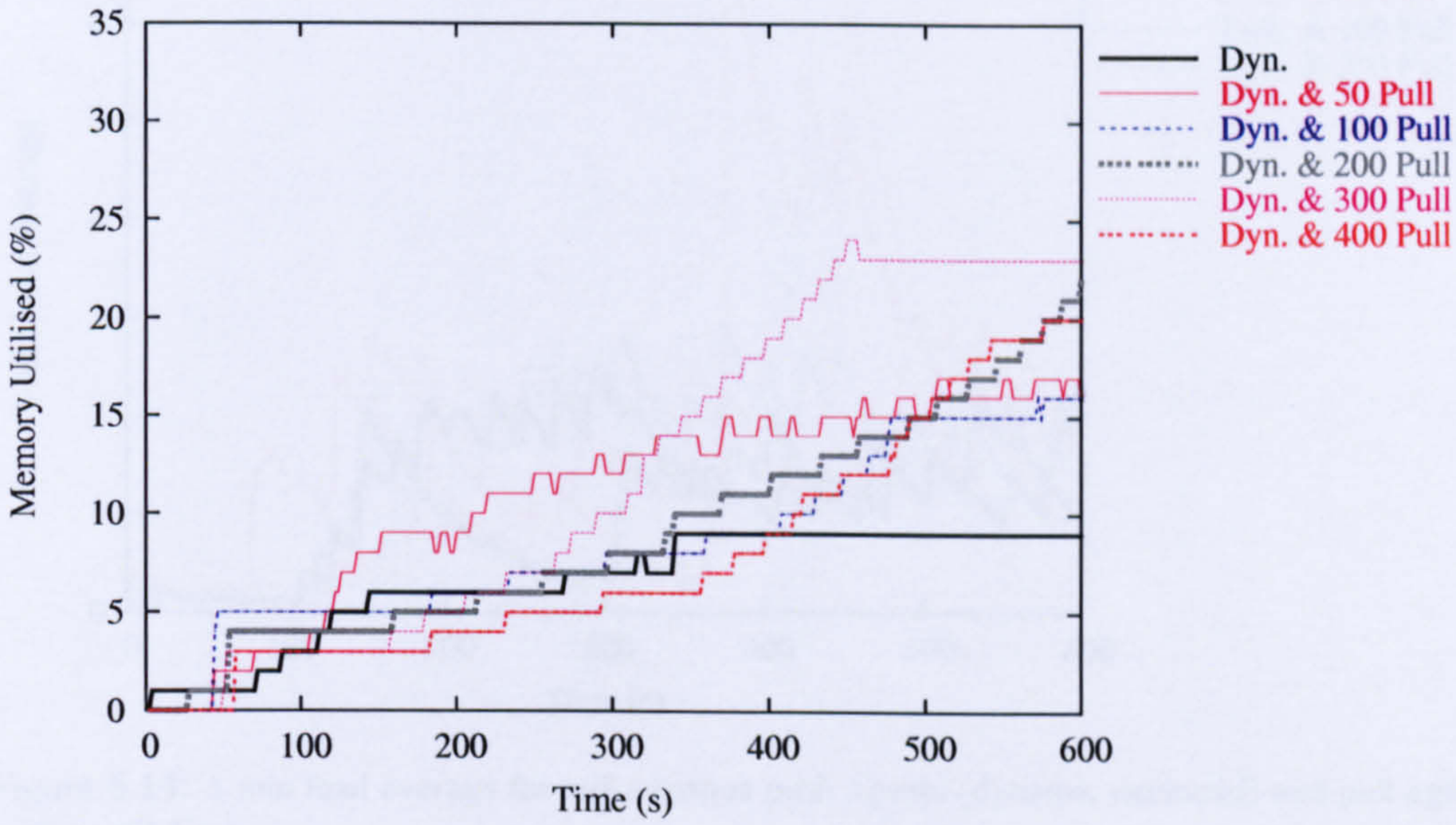

Figure 6.12: Percentage of memory utilised for self-adaptive push agents (dynamic workload) and pull agents with a 5 s wait time.

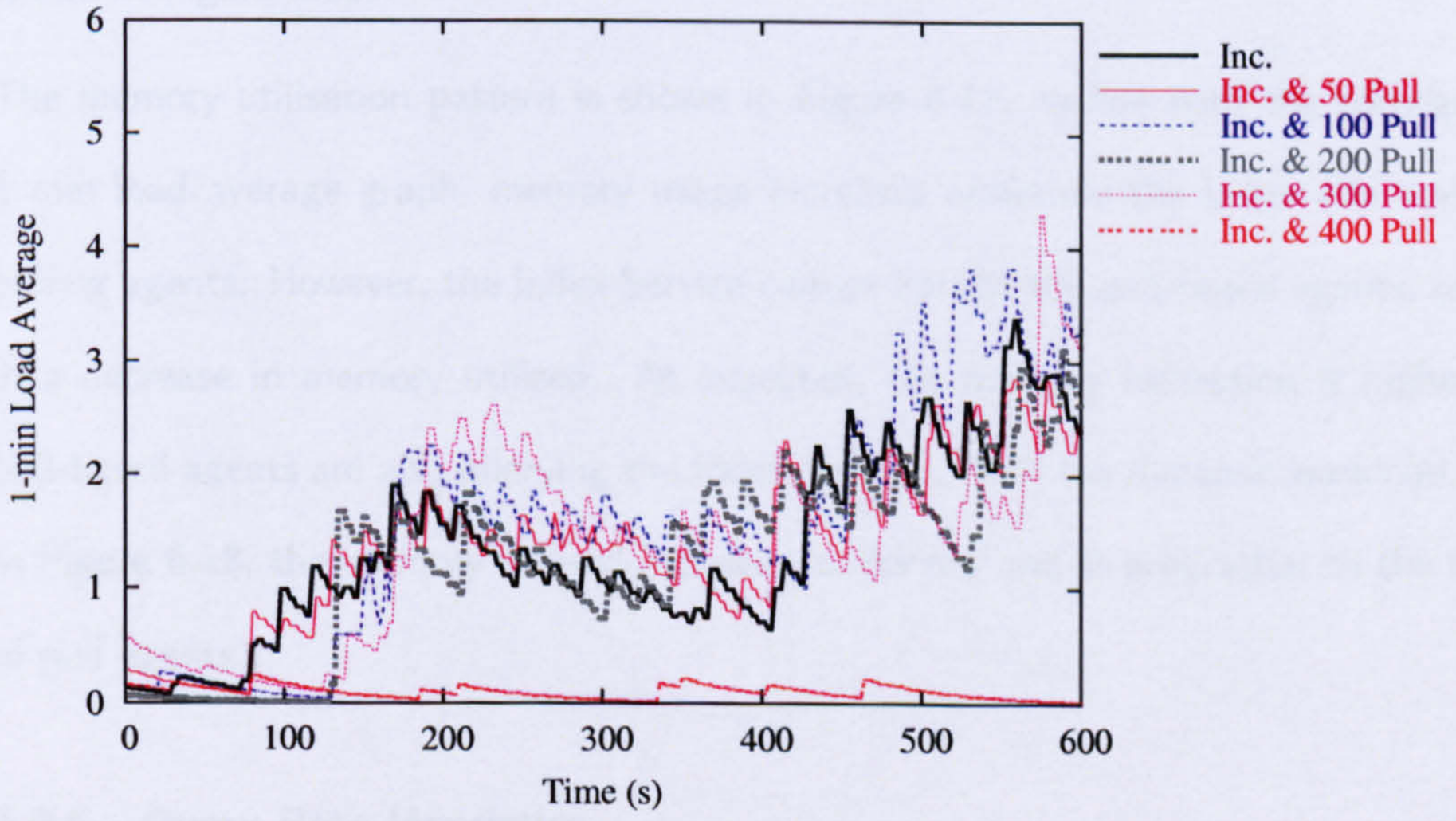

Figure 6.13: 1 min load average for self-adaptive push agents (increasing workload) and pull agents with a 30 s wait time.

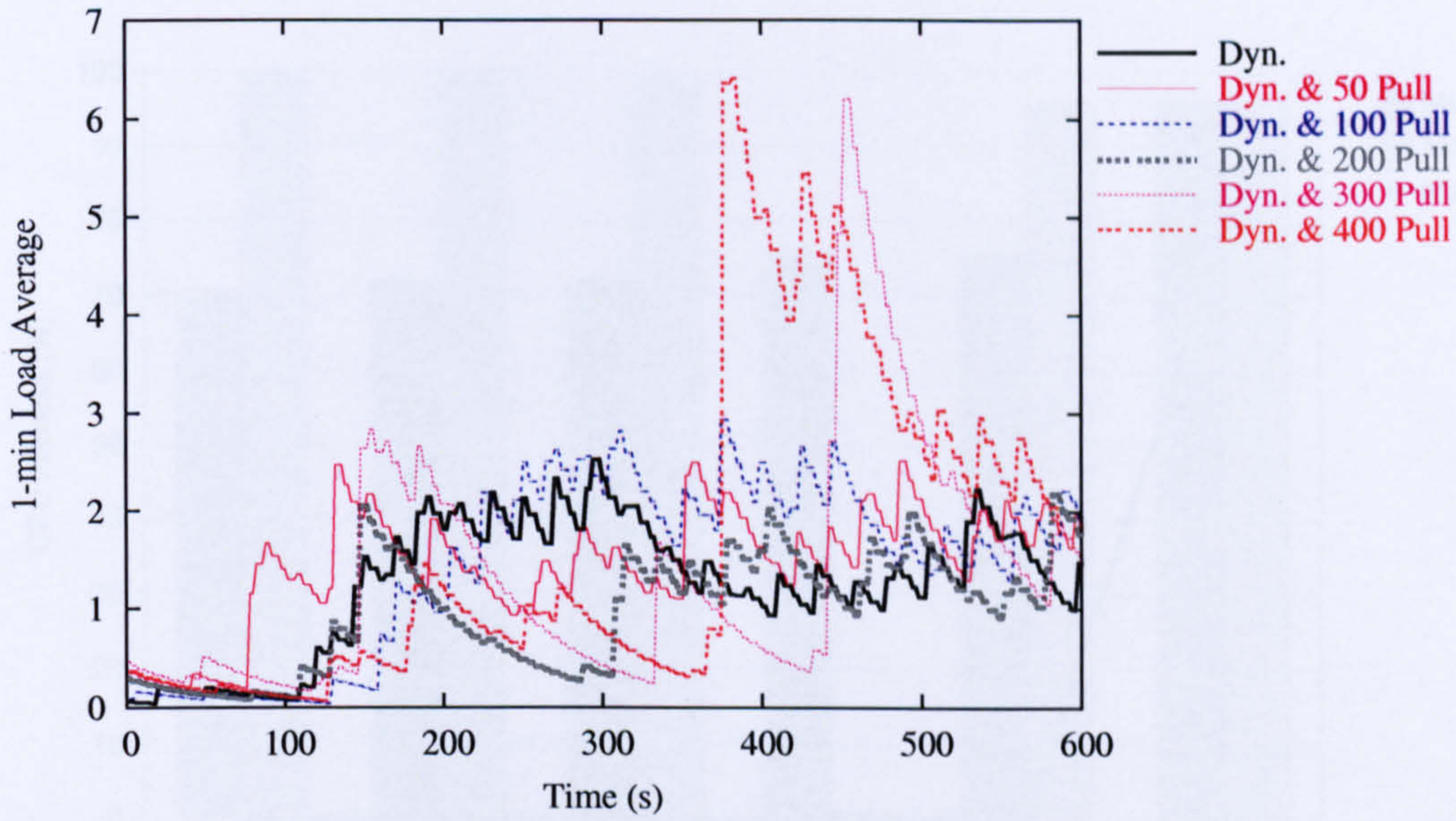

Figure 6.14: 1 min load average for self-adaptive push agents (dynamic workload) and pull agents with a 30 s wait time.

time experiment results (as shown by the mode). The graph also indicates that CPU utilisation is uniform across the various scenarios, but it increases by around 5% when there are 400 pull-based agents. Similar results occur with the dynamic workload, as shown in Figure 6.16.

The memory utilisation pattern is shown in Figure 6.17. In line with the corresponding 1 min load average graph, memory usage increases uniformly the larger the number of pulling agents. However, the Index Service cannot handle 400 pull-based agents, resulting in a decrease in memory utilised. As expected, the memory utilisation is higher when pull-based agents are also querying the Index Service. With the dynamic workload, shown in Figure 6.18, the memory utilised increases uniformly and in proportion to the number of pull agents.

6.3.5 Query Rate Heuristics

From an experimental point of view, the difference between a push-based sink and a pull-based client apart from their rate of arrival, is the nature of the query. While in

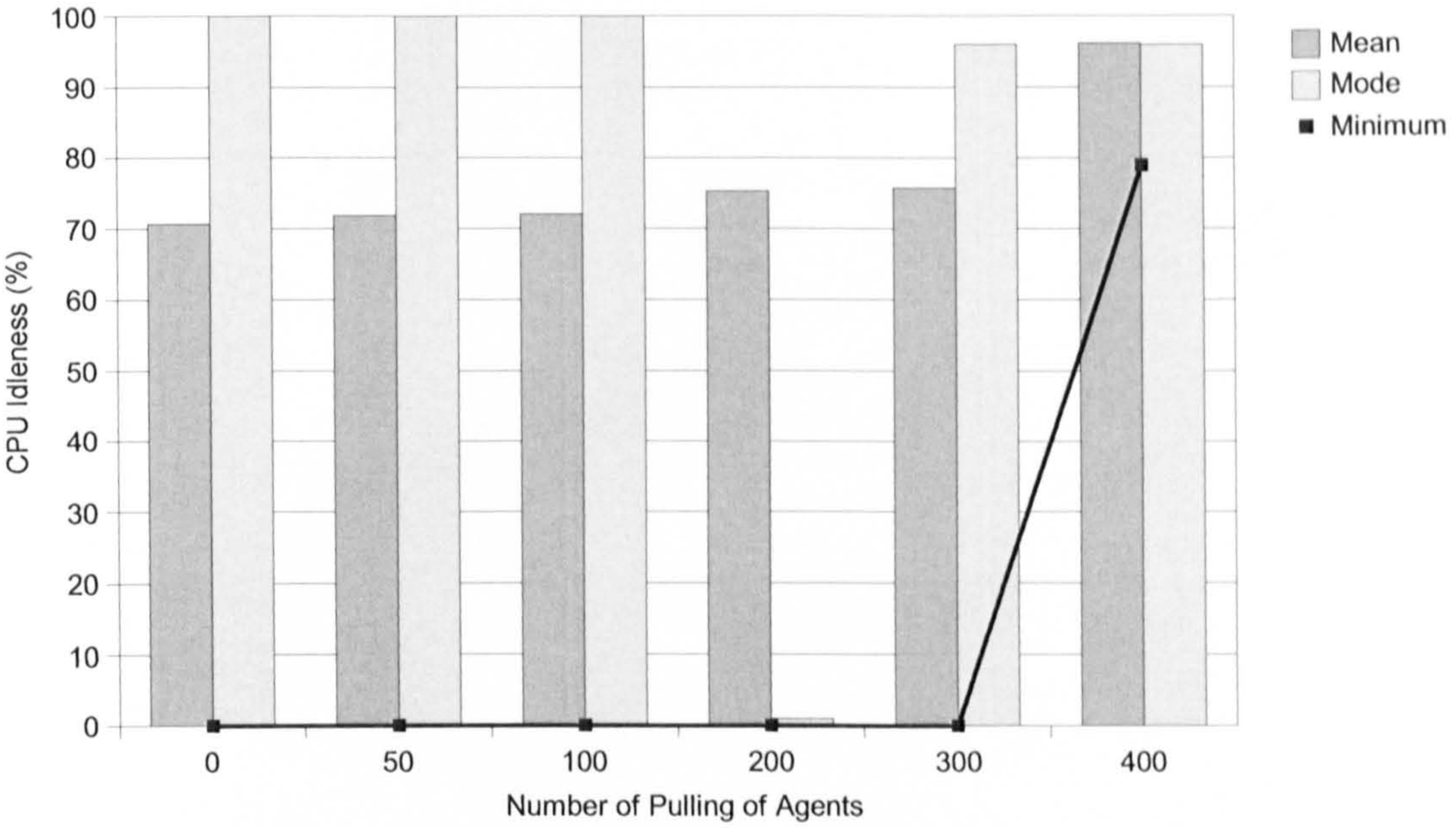

Figure 6.15: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents with a 30 s wait time.

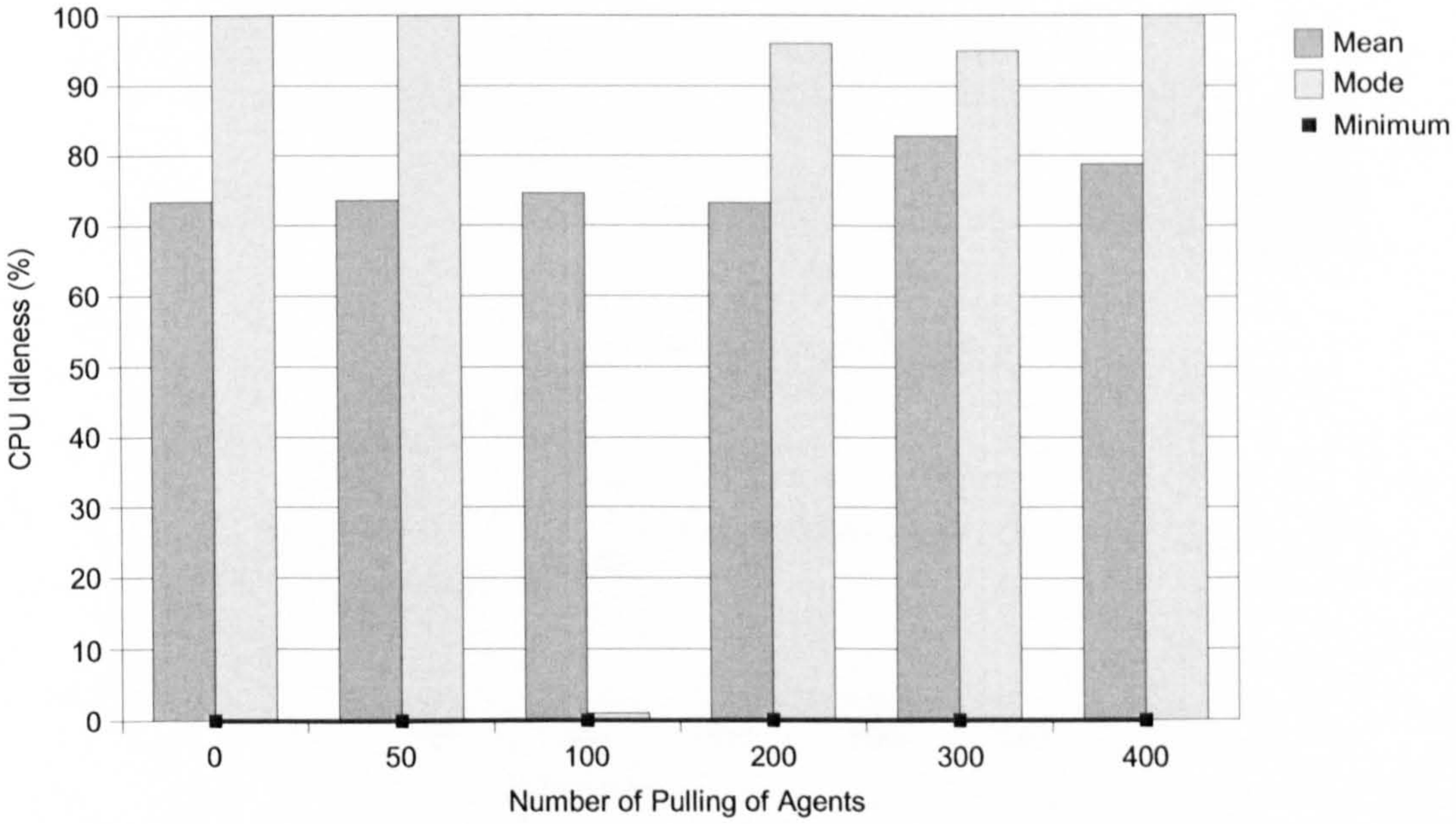

Figure 6.16: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents with a 30 s wait time.

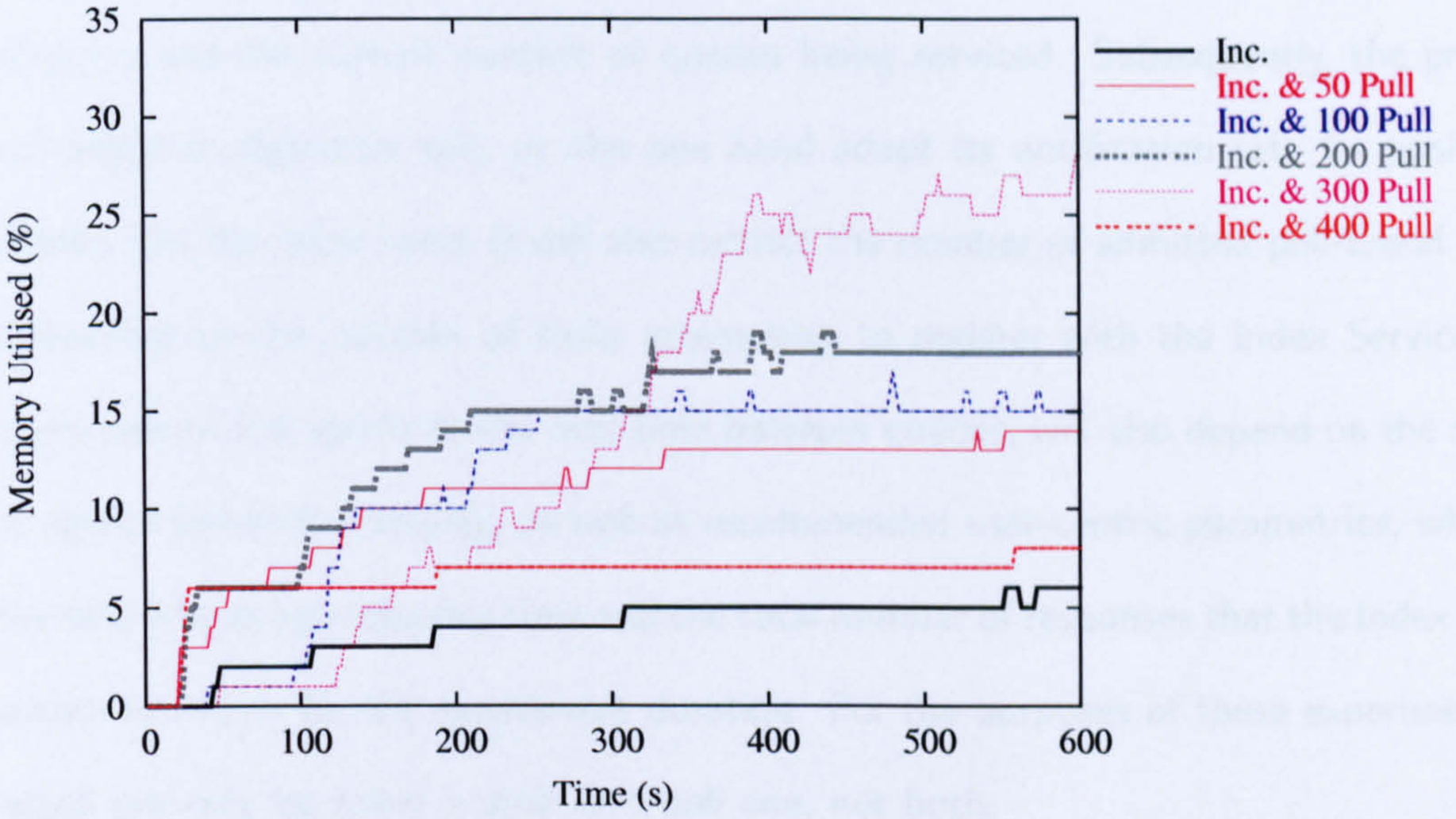

Figure 6.17: Percentage of memory utilised for self-adaptive push agents (increasing workload) and pull agents with a 30 s wait time.

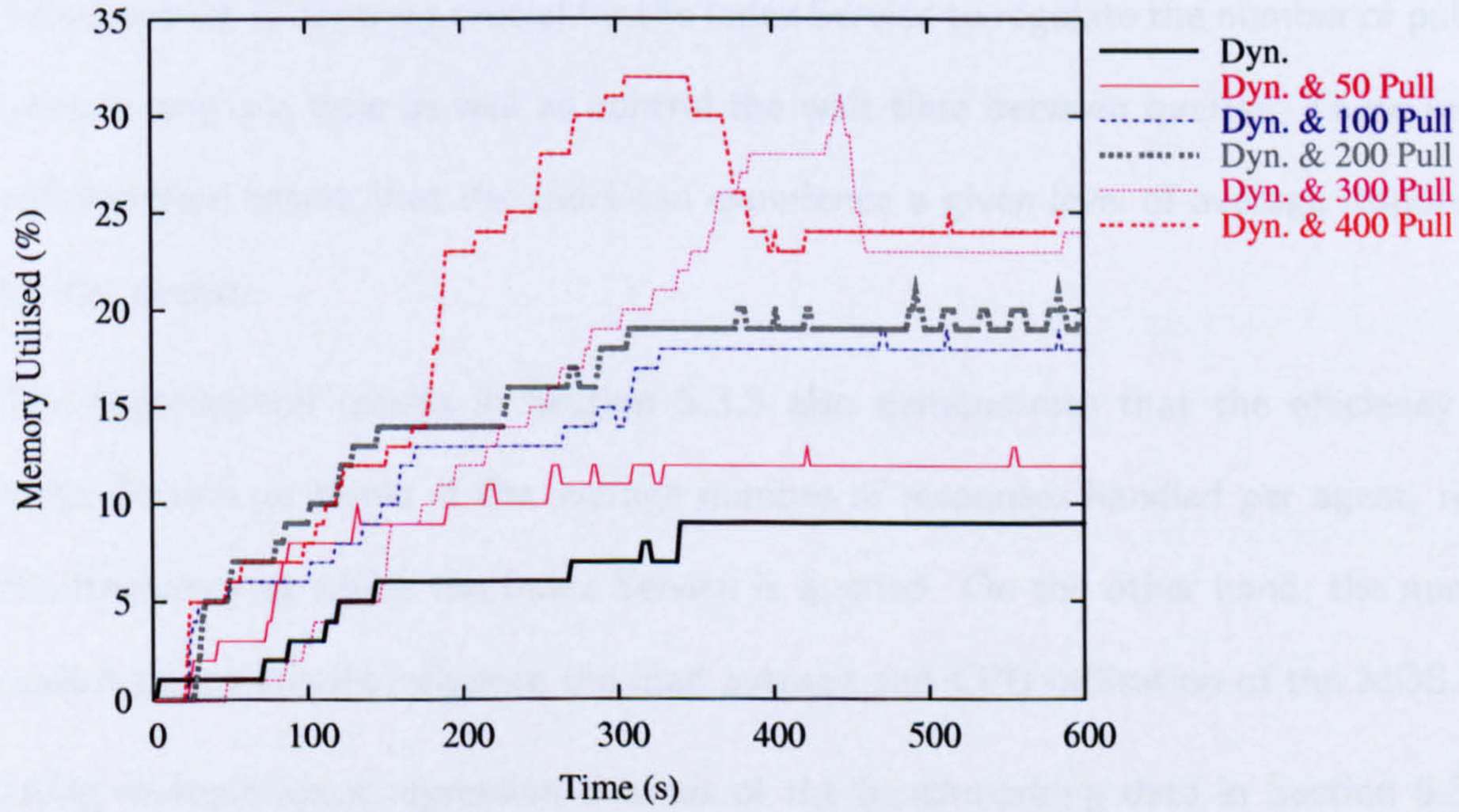

Figure 6.18: Percentage of memory utilised for self-adaptive push agents (dynamic workload) and pull agents with a 30 s wait time.

asynchronous queries, the control is imparted to the Index Service, in synchronous queries, the resources which the Index Service possesses for handling queries, depends on the rate of query and the current number of queries being serviced. Subsequently, the proposed self-adaptive algorithm will, on the one hand adapt its notification rate for push-based clients. On the other hand, it will also restrict the number of admitted pull-based clients, depending on the number of sinks attempting to register with the Index Service. The query rate of the agents or the wait time between queries, will also depend on the number of agents presently querying, as well as recommended user-centric parametrics, which are the desired average response time and the total number of responses that the Index Service wishes to return for the experiment duration. For the purposes of these experiments, an agent can only be either a push or a pull one, not both.

Another independent variable of pull-based queries is the time elapsed between each returned query and issuing the next query. Referring to Figure 5.12, it can be observed that as the number of agents which can be serviced in a synchronous manner increases, so does the average response time for obtaining the results of a query, the shorter the wait time. Subsequently, it becomes crucial for the Index Service to regulate the number of pull-based users at any one time as well as control the wait time between queries. These measures will therefore ensure that the users can experience a given level of average response time for the queries.

The experimental results in Section 5.3.3 also demonstrate that the efficiency of the Index Service, in terms of the average number of responses handled per agent, relies on the frequency at which the Index Service is queried. On the other hand, the number of asynchronous queries influence the load average and CPU utilisation of the MDS.

Using multiple linear regression analysis of the benchmarking data in Section 6.3.2, the equation below is obtained:

$$Wait = a + bAgents + cResponseTime + dTotalResponses \tag{6.1}$$

where

$Wait$ is the wait time between receiving query results and issuing the next query

$Agents$ is the current number of pull-based agents

$ResponseTime$ is the desired average response time per query

$TotalResponses$ is the desired total number of responses for a given period

A description of the benchmarking data sets used is given below:

- **D1** Small sets of data with small wait times, using the average number of responses as the user-centric parametric;
- **D2** Large sets of data with small wait times, using the average number of responses as the user-centric parametric;
- **D3** Small sets of data with large wait times, using the average number of responses as the user-centric parametric;
- **D4** Small sets of data with large wait times, using the total number of responses as the user-centric parametric;
- **D5** Large sets of data with small wait times, using the total number of responses as the user-centric parametric.

The benchmarking data has been analysed and their multiple regression calculated. Their 95% confidence interval and RMSE (root mean square error) from Equation 6.2 are also calculated, and results show that D4 and D5 display the smallest differences in their means and that of the actual observed data, as shown in Figure 6.19. The method giving

the least RMSE (root mean square error) from Equation 6.2 is chosen, from which the values of the constants a to d are obtained. The data set chosen is therefore D4, with an RMSE value of 4.8. The corresponding confidence interval for D4 is comparatively small, therefore giving a satisfactory prediction.

$$RMSE = \sqrt{\frac{1}{N}\sum_{x}(O_x - C_x)^2} \tag{6.2}$$

where

N is the data sample set size

O is an observation value

C is a calculated value via multiple linear regression

The values of the constants are:

$a = 23.742$

$b = 0.110$

$c =$ -0.975

$d =$ -0.004 to 3 decimal places

6.3.6 Self-adaptive Pull Queries

The experiments in this sub-section involve pull-based agents alone, with their wait times being decided by the multiple linear regression equation 6.1. The client-side performance results are shown in the following graphs.

The self-adaptive pulling algorithm produces a fairly high number of responses per agent, which closely follows that of the 5 s wait time. This is shown in Figure 6.20. The average

	RMSE ± 95% Confidence Interval
D1	7.95 ± 5.94
D2	6.49 ± 3.94
D3	6.22 ± 4.73
D4	4.80 ± 4.86
D5	5.07 ± 4.05

Figure 6.19: Comparing the different data sets in terms of their 95% confidence interval and RMSE.

query response time is also improved when compared with the 1 s and 5 s wait time scenarios, as shown in Figure 6.21. Self-adaptation gives an average query response time of 66 s, corresponding to 1000 pulling agents. Additionally, Figure 6.22 indicates that the self-adaptive algorithm gives a total number of successful responses comparable with the 5 s wait time scenario. In brief, it can be observed that the self-adaptive algorithm produces reliable performance metrics.

6.3.7 Both Self-adaptive Push and Pull Queries

Experiments in this sub-section involve push-based agents which are represented by *increasing* and *dynamic* workloads, as well as pull-based agents whose wait time is calculated with 6.1. The same self-adaptive algorithm is also applied to the push-based agents. Experimental results obtained from a set of repeated experiments, are shown in the following graphs.

As can be seen in Figure 6.23, there is a slight improvement in the 1 min load average since load is existent for all of the experiment scenarios. Figure 6.32 confirms that a small

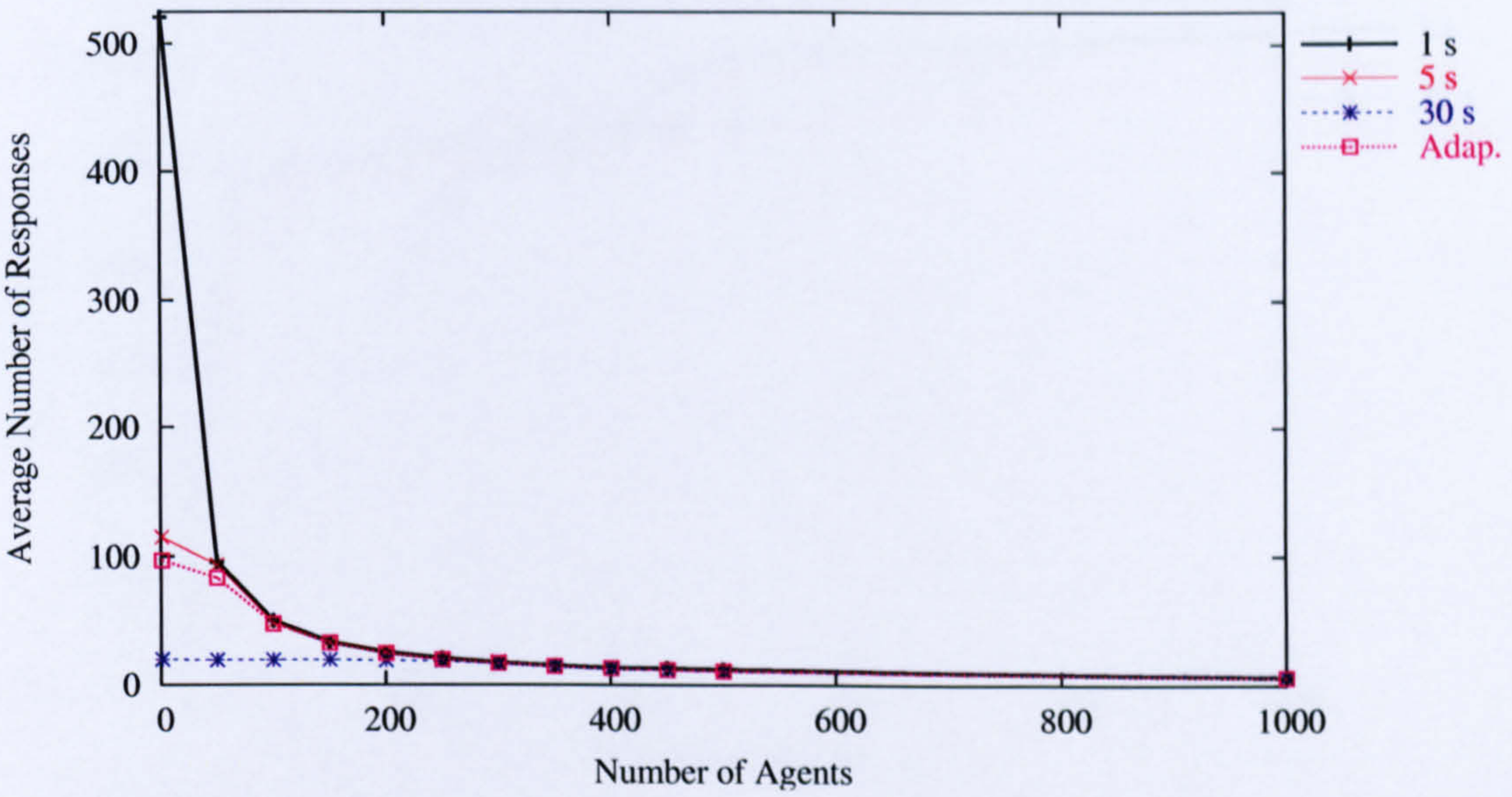

Figure 6.20: Average number of responses per agent with an increasing number of pull-based agents.

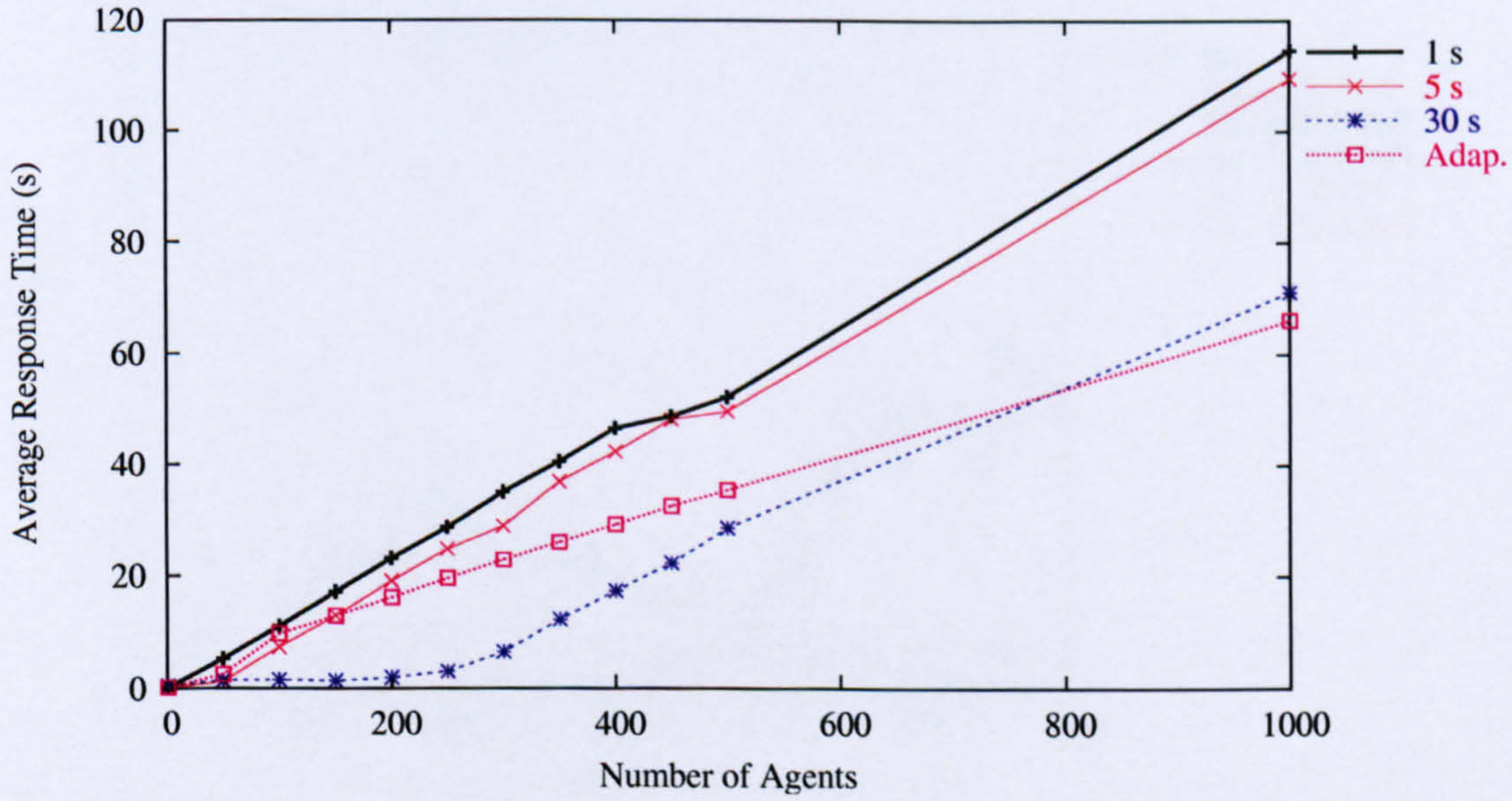

Figure 6.21: Average query response time with an increasing number of pull-based agents.

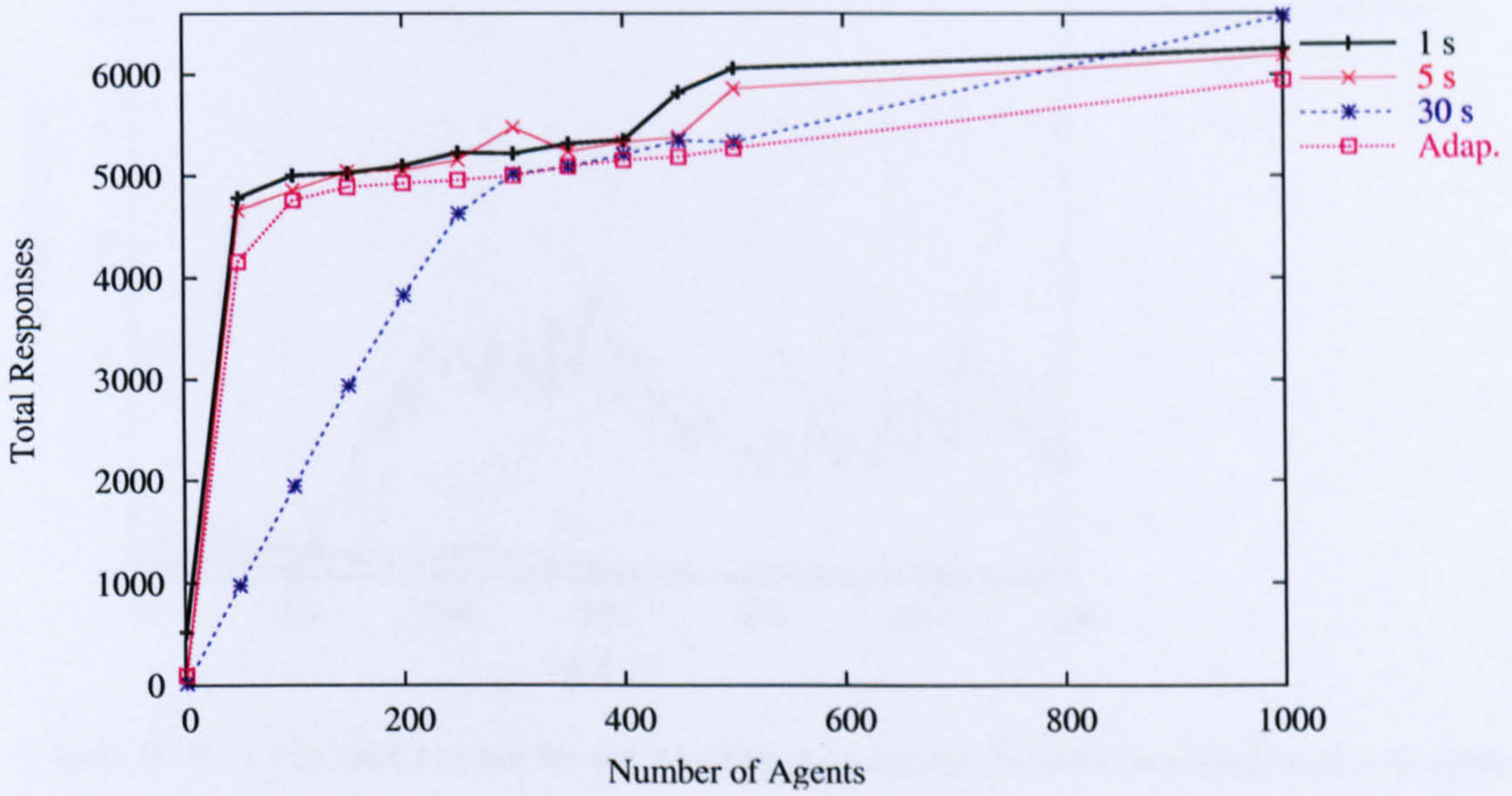

Figure 6.22: Total number of responses for the experiment duration, with an increasing number of pull-based agents.

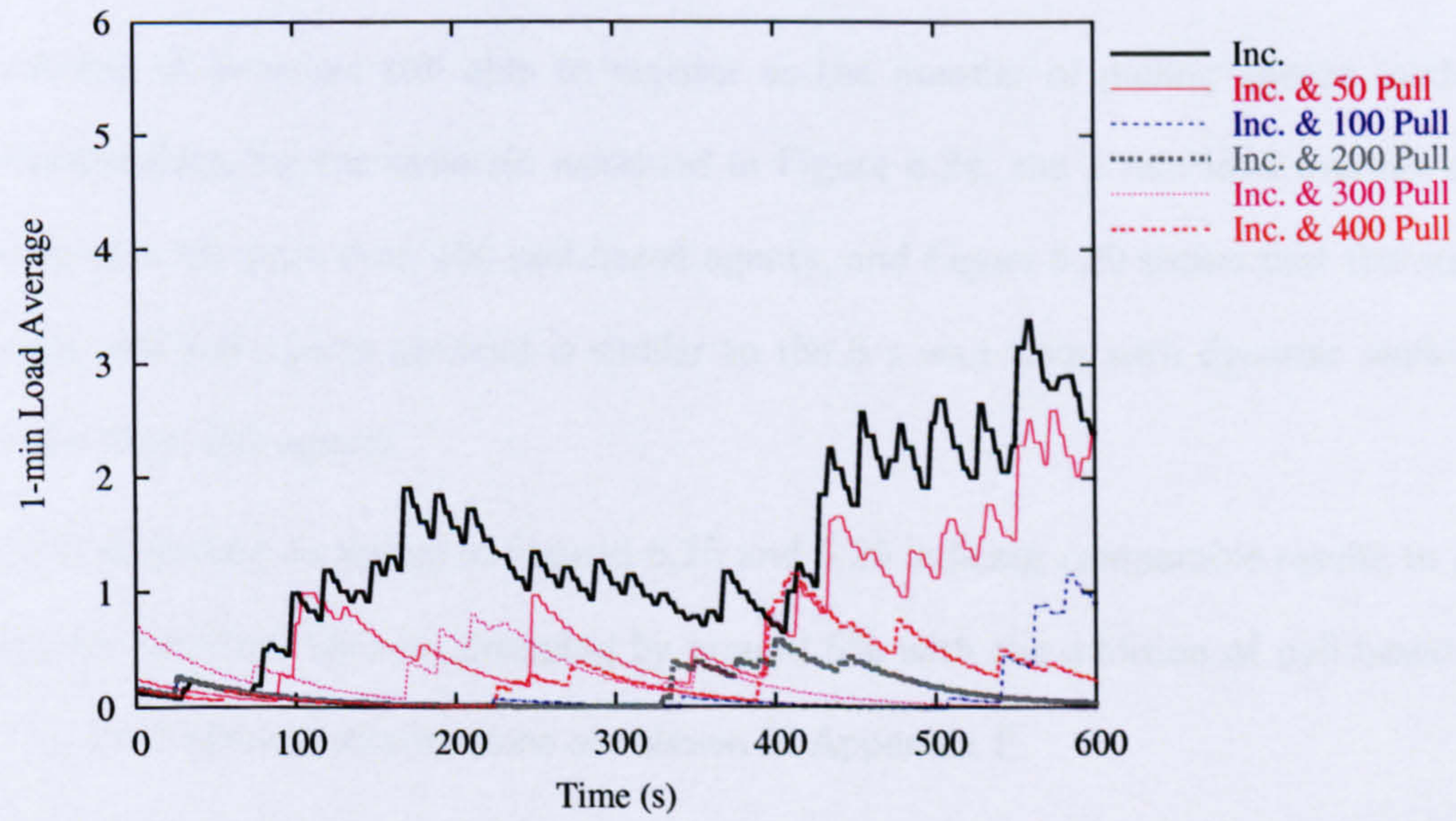

Figure 6.23: 1 min load average for self-adaptive push agents (increasing workload) and pull agents.

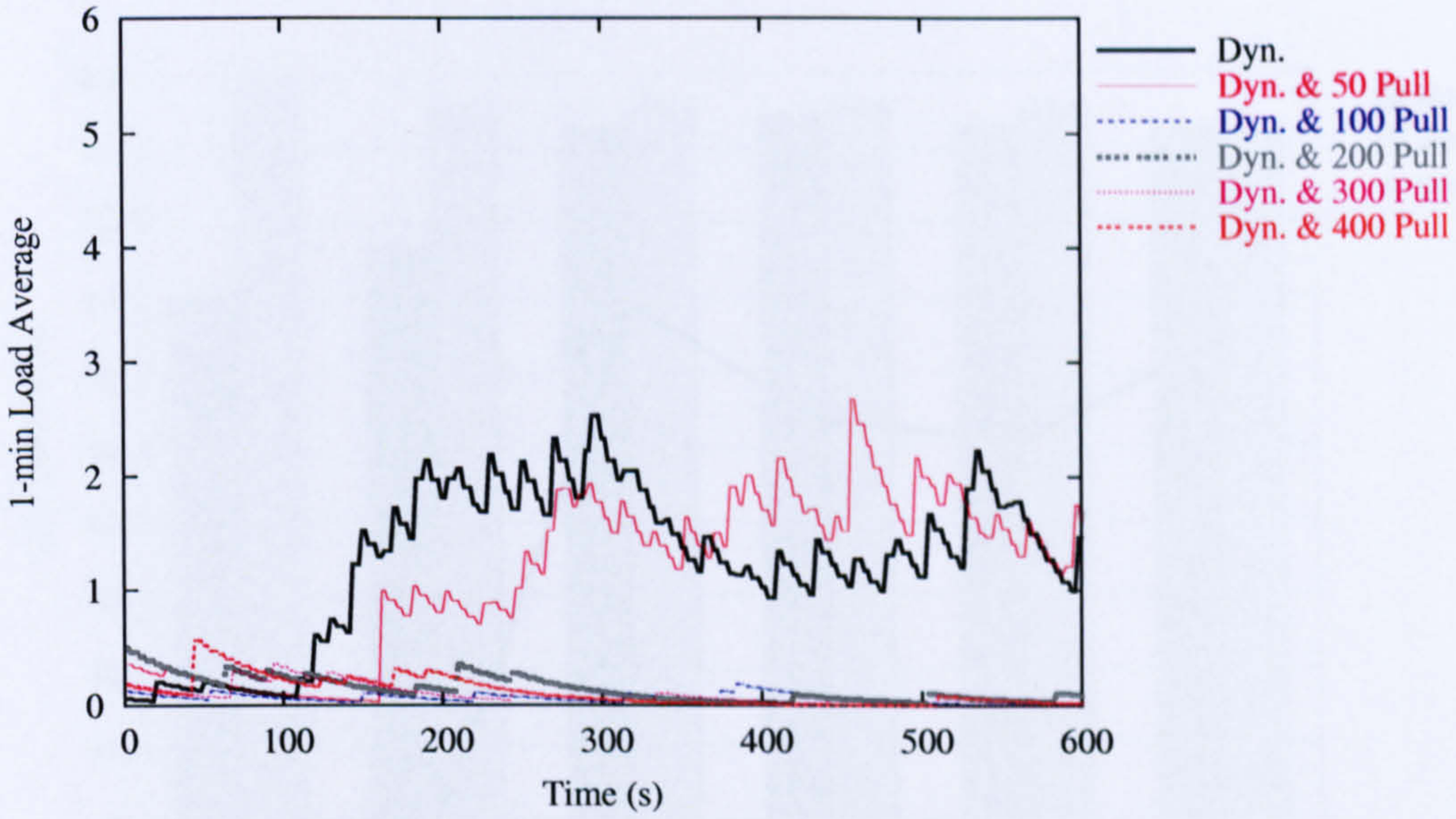

Figure 6.24: 1 min load average for self-adaptive push agents (dynamic workload) and pull agents.

number of sinks are still able to register as the number of pulling agents reaches 400. Nevertheless, for the dynamic workload in Figure 6.24, the 1 min load average drops to near zero for more than 100 pull-based agents, and Figure 6.29 shows that the number of push and pull agents serviced is similar to the 5 s wait time with dynamic workload, for more than 200 agents.

CPU utilisation as shown in Figures 6.25 and 6.26 indicate comparable results to previous results, with the idleness dropping by around 5% with the addition of pull-based agents. The CPU idleness distributions are shown in Appendix E.

The memory utilisation for both the increasing and dynamic workloads, as shown in Figures 6.27 and 6.28, indicate that the different scenarios use relatively low memory. The highest usage at around 25%, is with sinks combined with 50 pulling agents.

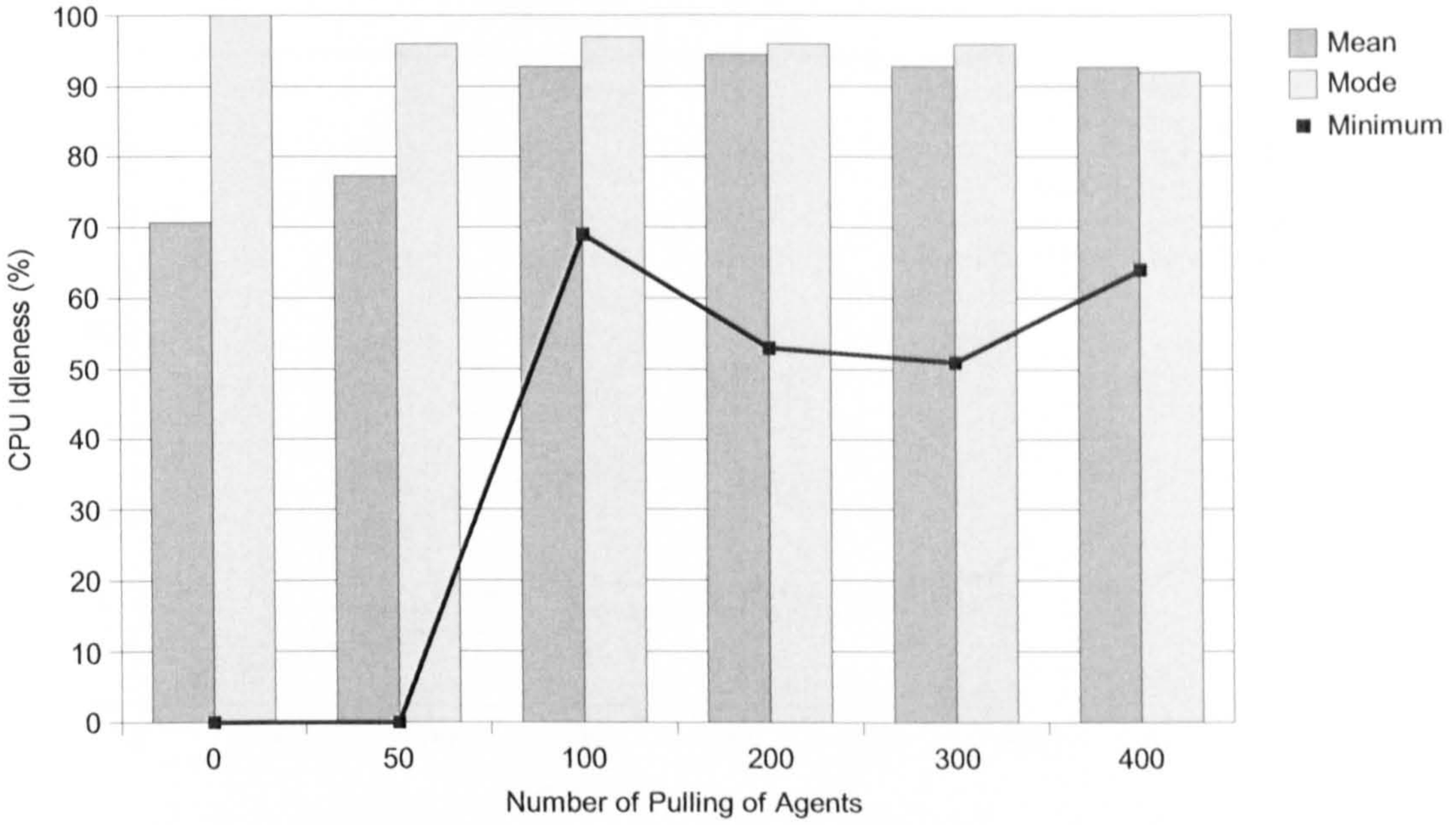

Figure 6.25: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents.

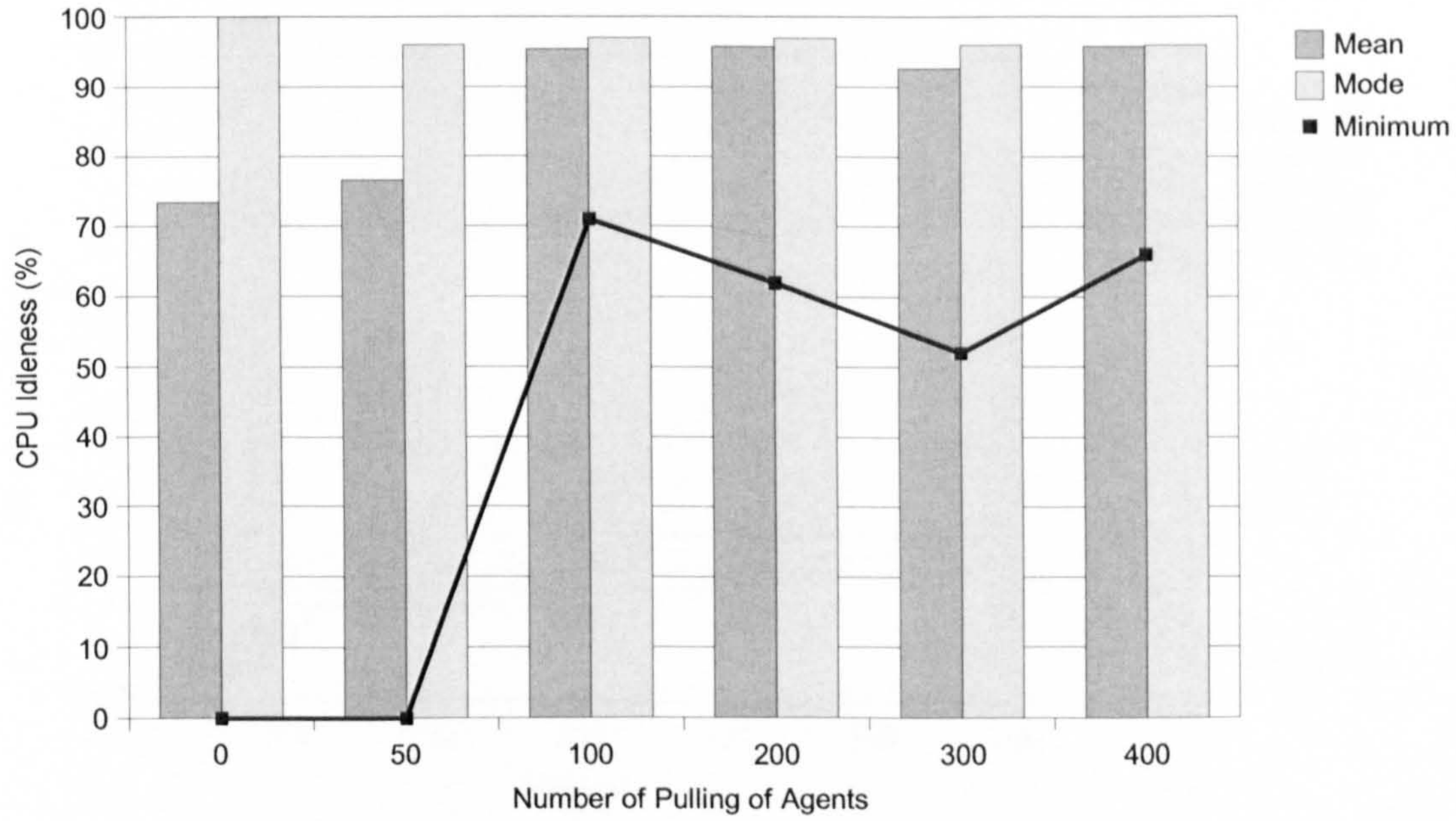

Figure 6.26: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents.

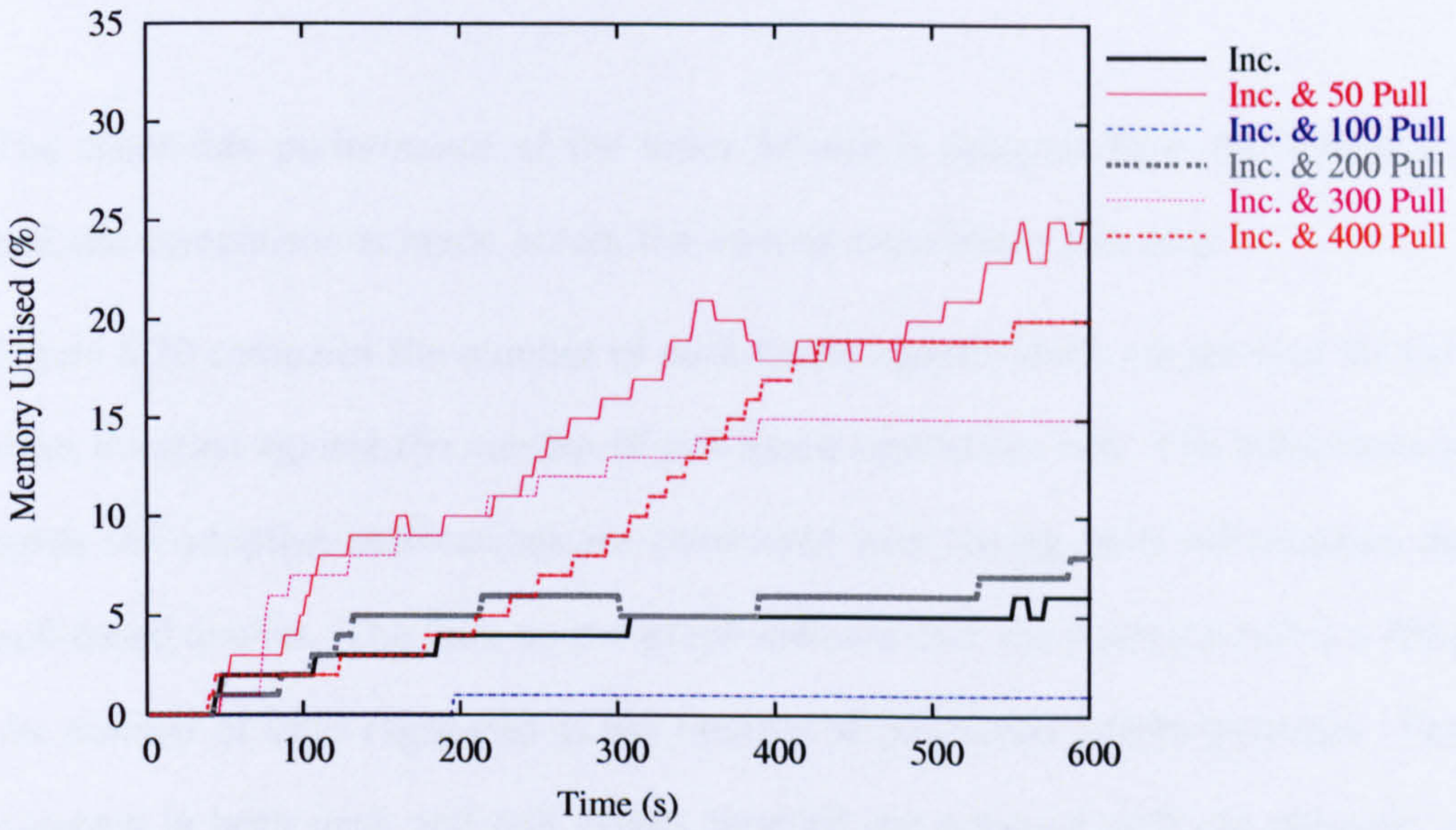

Figure 6.27: Percentage of memory utilised for self-adaptive push agents (increasing workload) and pull agents.

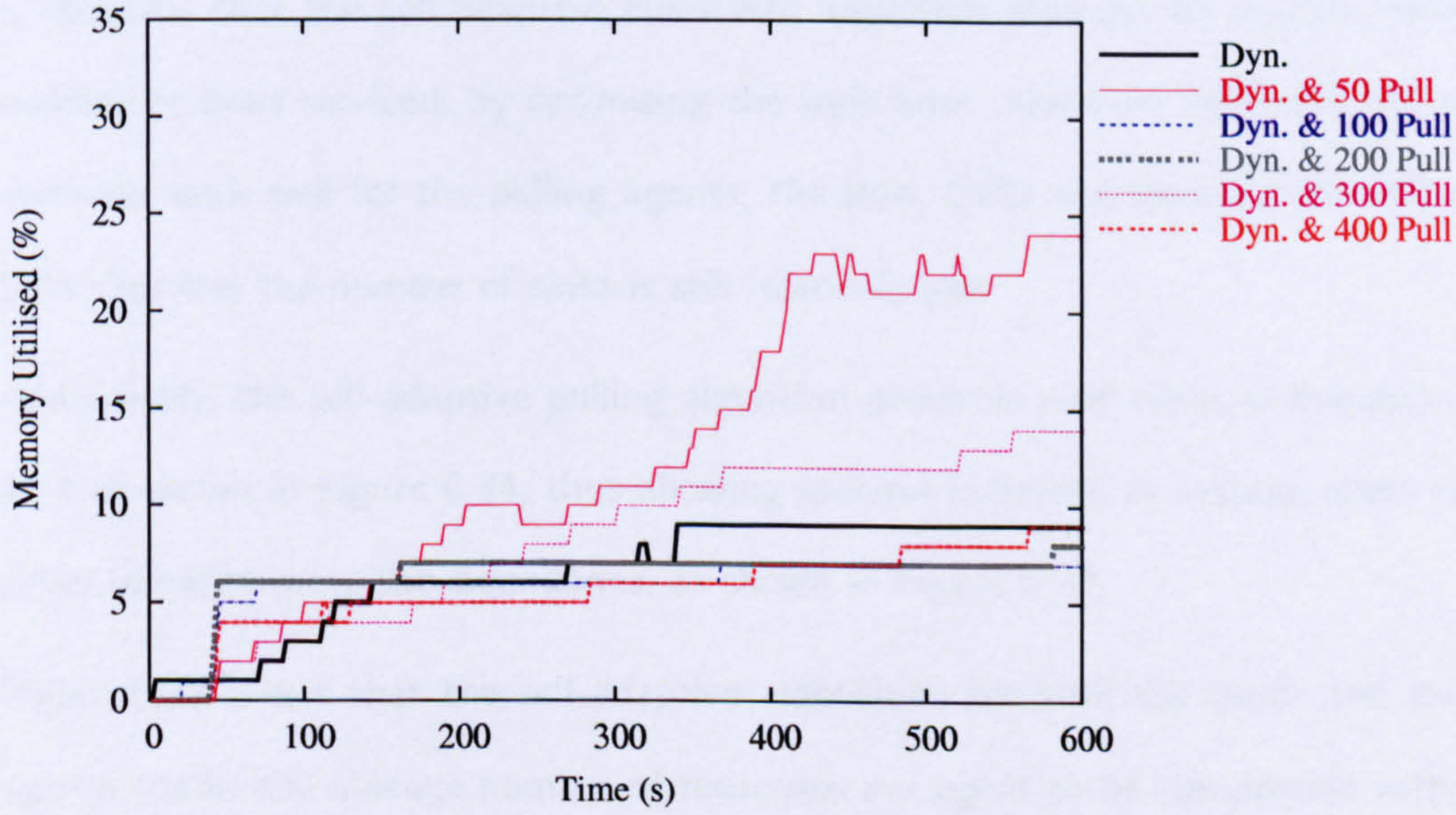

Figure 6.28: Percentage of memory utilised for self-adaptive push agents (dynamic workload) and pull agents.

6.3.8 Comparison of Client-Side Parametrics with Self-adaptive Push and Pull Queries

The client-side performance of the Index Service is analysed from the following graphs, and the comparison is made across the various experiment scenarios.

Figure 6.29 compares the number of push-based agents which are serviced for the experiment duration against the number of pull-based agents serviced. The differences in having solely self-adaptive notifications are contrasted with having both self-adaptive push- and pull-based queries. The lines on the graph indicate that the general trend is a reduction in the number of sinks registered as the number of pull-based agents increases. The largest numbers in both push and pull agents serviced are achieved with the dynamic workload and a 30 s wait time for the synchronous agents. The Index Service can therefore simultaneously service 400 pulling agents and around 350 push agents. Moreover, the dynamic workload with the 5 s wait time gives the largest drop in the number of sinks being registered as the number of pull-based agents increases. Consequently, from the graphs, it is observed that the self-adaptive pull-based algorithm provides an improvement in the number of sinks serviced, by optimising the wait time. However, while the self-adaptive methods work well for the pulling agents, the load, CPU and memory utilisation of the Index Service, the number of sinks is still relatively low.

Additionally, the self-adaptive pulling algorithm produces wait times of between 3 s and 19 s as shown in Figure 6.34, thus allowing uniform increases in average query response times between these two boundaries, as shown in Figure 6.30.

Figure 6.31 shows that the self-adaptive algorithms for both the push- and pull-based agents enable the average number of responses per agent to be comparable with the 5 s wait time experiment scenarios. Whilst the average responses per agent with the 30 s wait time hardly reaches 20 responses per agent, the self-adaptive algorithms allow the average number of responses to decay exponentially. Figure 6.32 shows the numbers of push- and

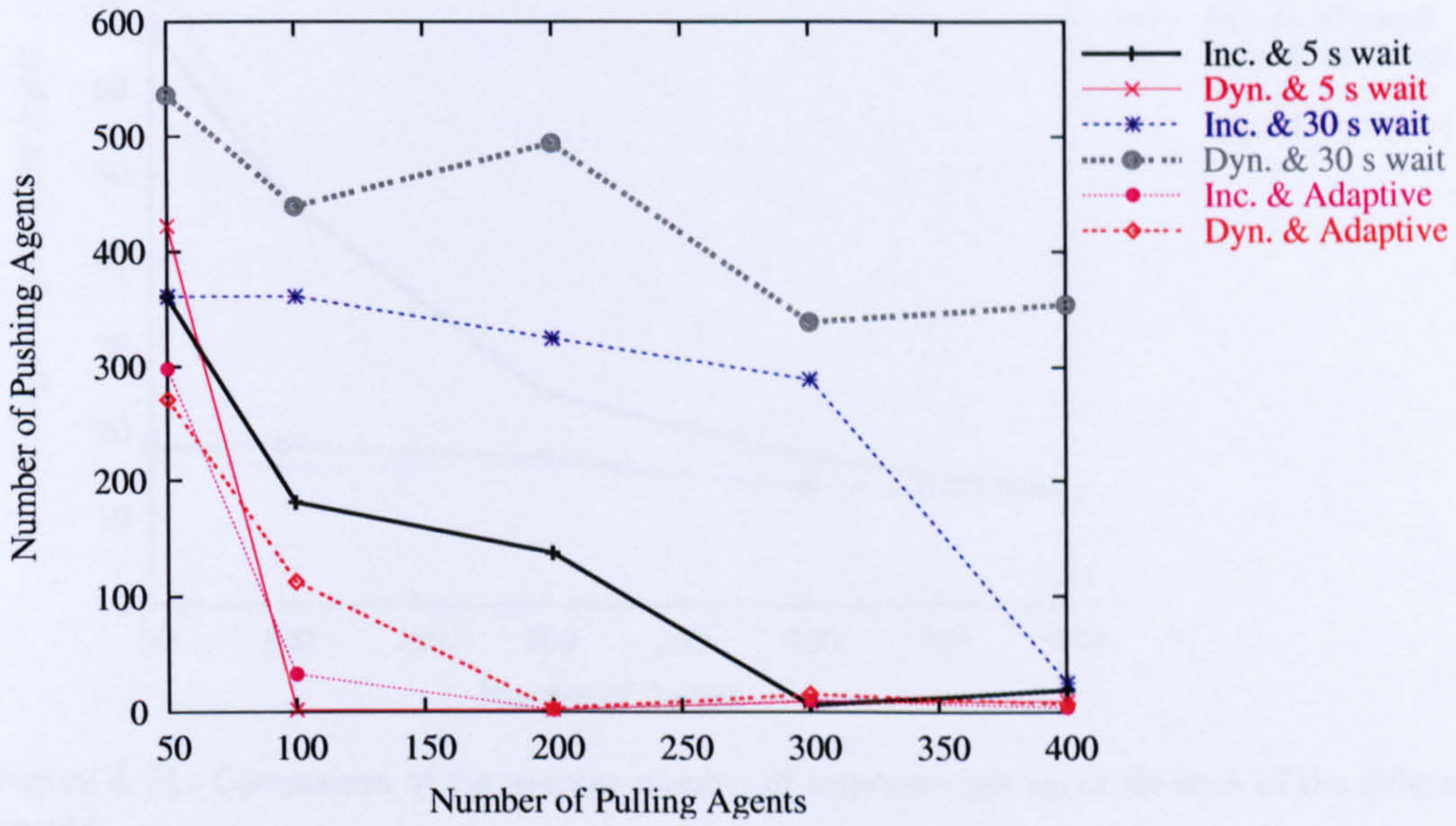

Figure 6.29: Comparison of the number of push- and pull-based agents serviced for each of the different experiments.

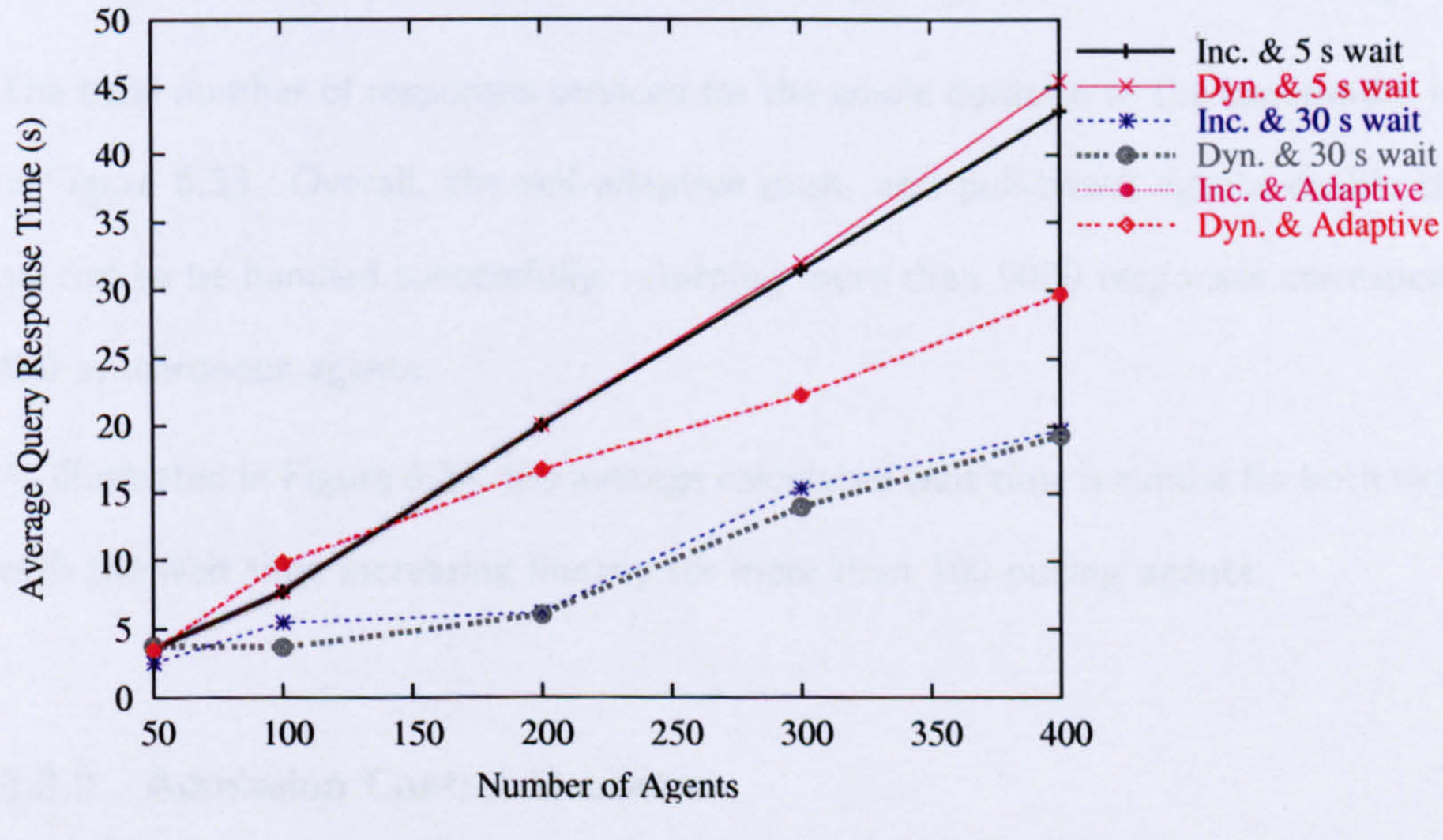

Figure 6.30: Comparison of the average query response time for each of the different experiments.

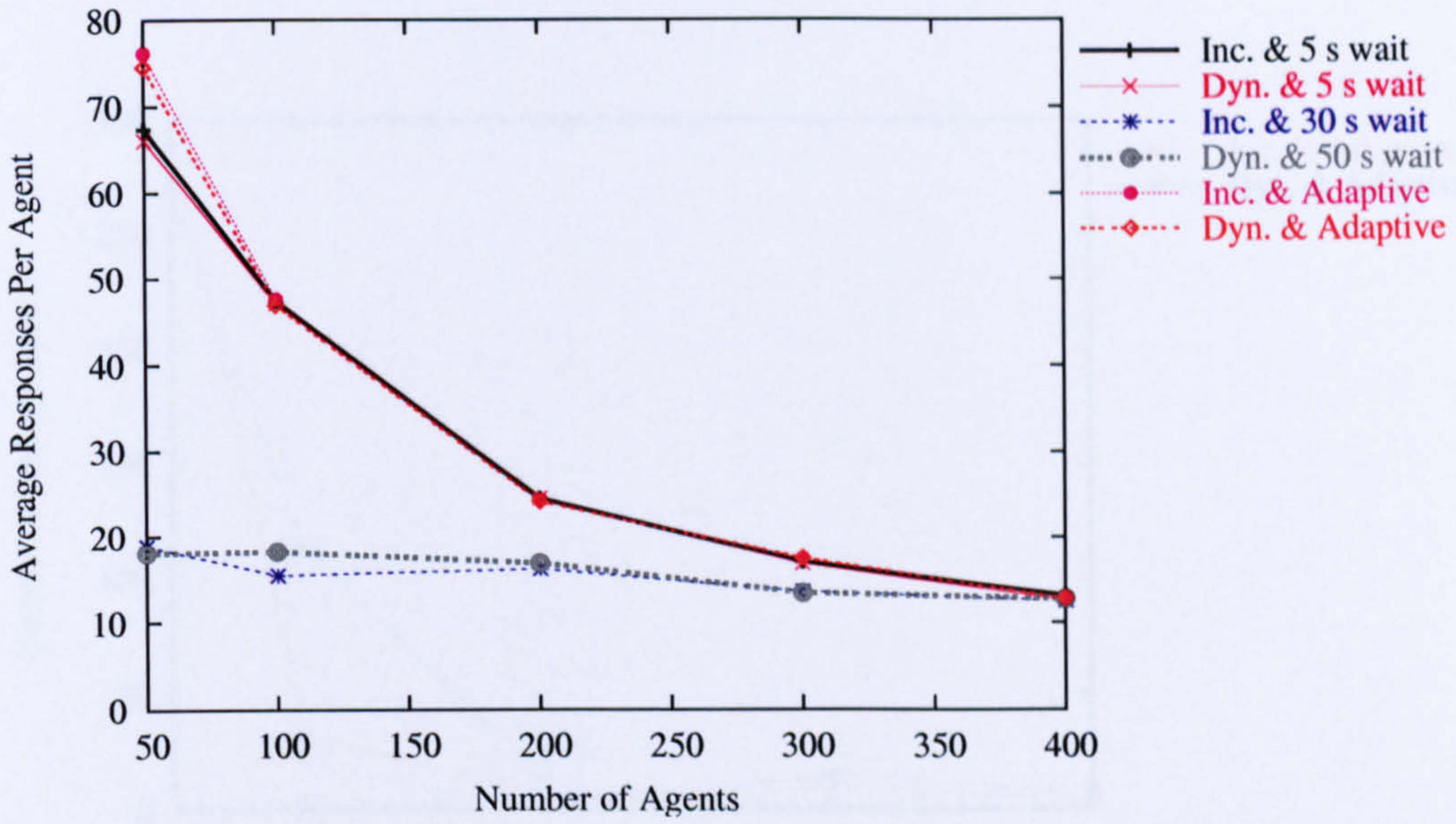

Figure 6.31: Comparison of the average number of responses per agent for each of the different experiments.

pull-based agents serviced with the self-adaptive algorithms alone. The graphs show that the number of sinks start to decay significantly when there are more than 100 agents querying the Index Service synchronously. Furthermore, the dynamic workload produces a better balance between the number of push and pull agents, than the increasing workload.

The total number of responses serviced for the whole duration of the experiment is shown in Figure 6.33. Overall, the self-adaptive push- and pull-based agents enable the most queries to be handled successfully, returning more than 5000 responses corresponding to 400 synchronous agents.

As illustrated in Figure 6.34, the average calculated wait time is similar for both workloads, with the wait time increasing linearly for more than 100 pulling agents.

6.3.9 Admission Control Heuristics

The introduction of the self-adaptive algorithms to both the push and pull query produces an improvement for the synchronous agents. However, the number of sinks able to register

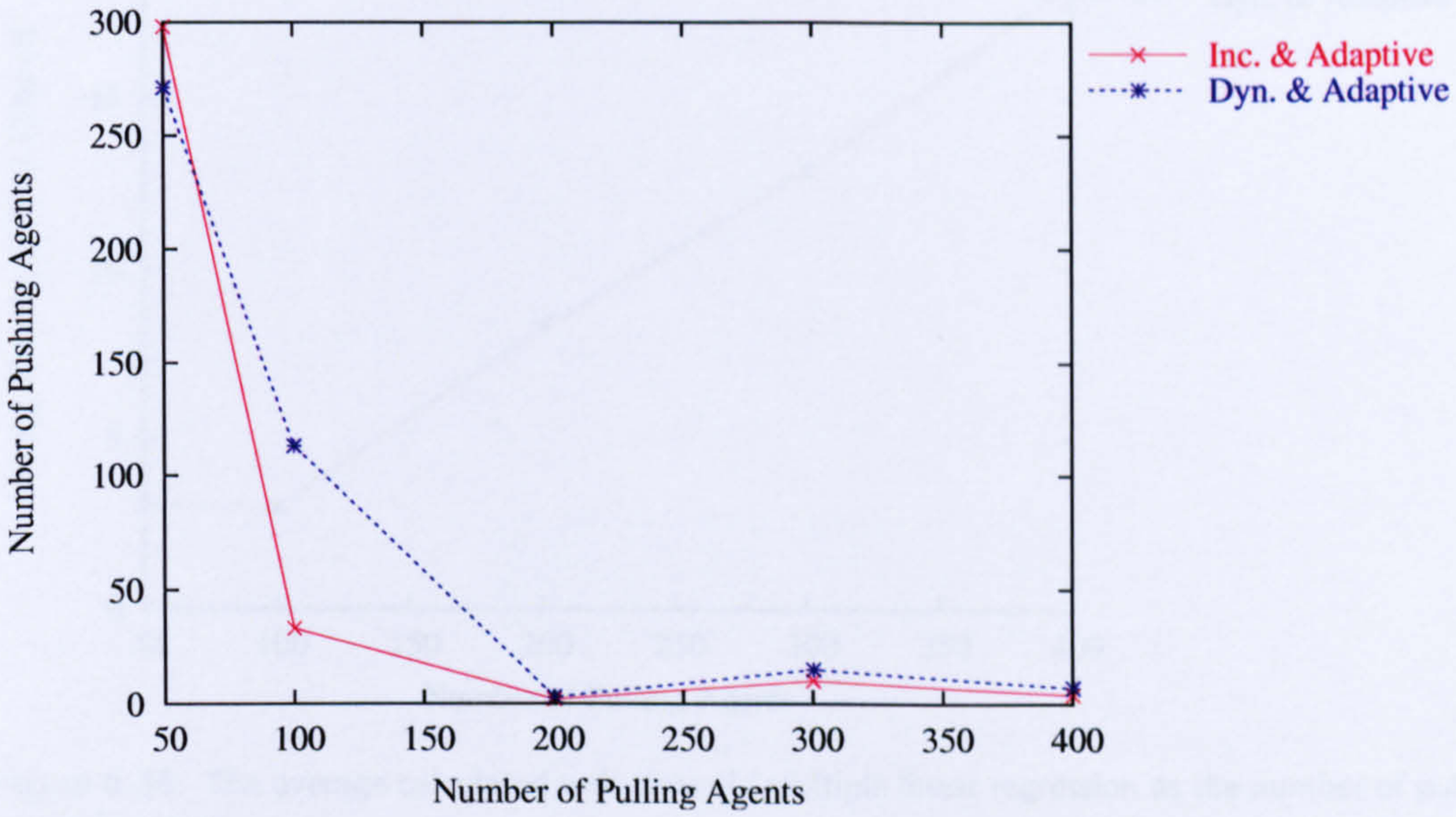

Figure 6.32: Comparison of the increasing and dynamic workloads for self-adaptive push- and pull-based agents.

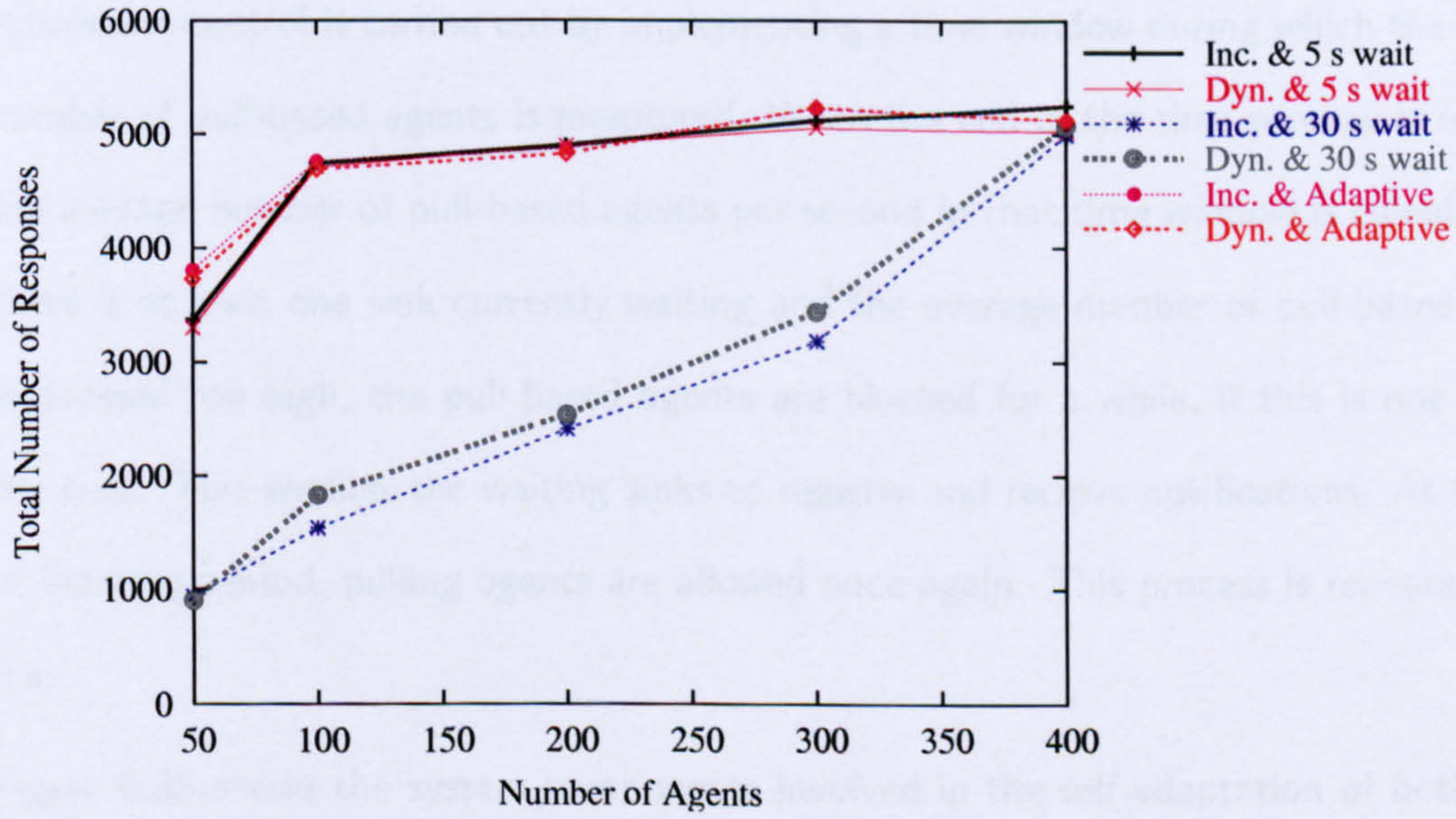

Figure 6.33: Comparison of the total number of responses for the duration of the experiment for each of the different experiments.

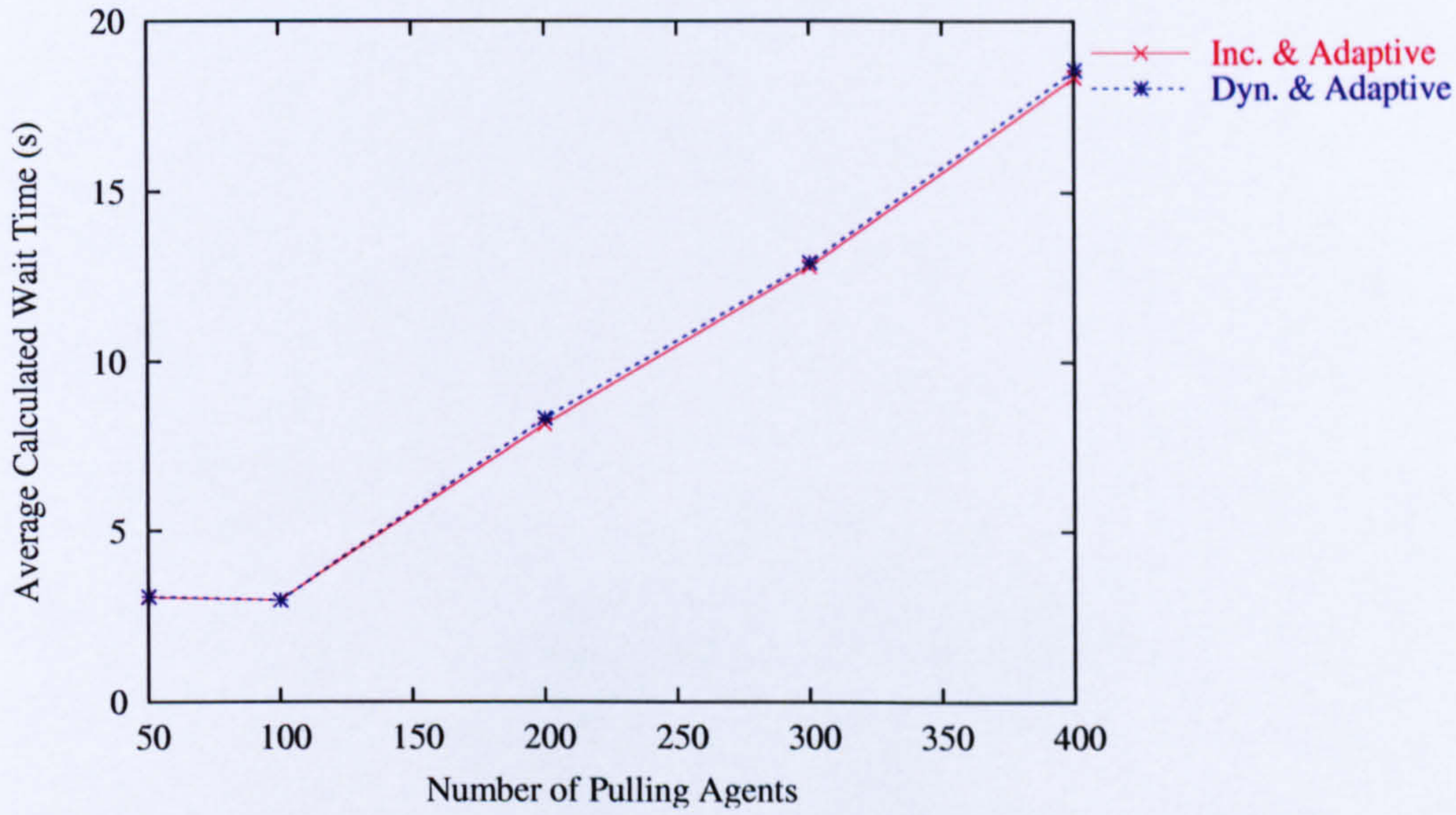

Figure 6.34: The average calculated wait time via multiple linear regression as the number of pull-based agents increases.

with the Index Service is still low. Subsequently, an admission control algorithm is added to GridAdapt to limit the synchronous queries for a certain length of time to allow more sinks to be registered.

Admission control is carried out by implementing a time window during which the current number of pull-based agents is monitored. When the end of the time window is reached, the average number of pull-based agents per second in that time window is calculated. If there is at least one sink currently waiting and the average number of pull-based agents is deemed too high, the pull-based agents are blocked for a while, if this is not already the case. This enables the waiting sinks to register and receive notifications. At the end of blocking period, pulling agents are allowed once again. This process is repeated every 2 s.

Figure 6.35 shows the system components involved in the self-adaptation of both push- and pull-based agents. The Index Service, push- and pull-based agents, and the self-adaptive and admission controller part of GridAdapt are physically located on different hosts, to minimise the impact on load each one has on the other. The monitor on the

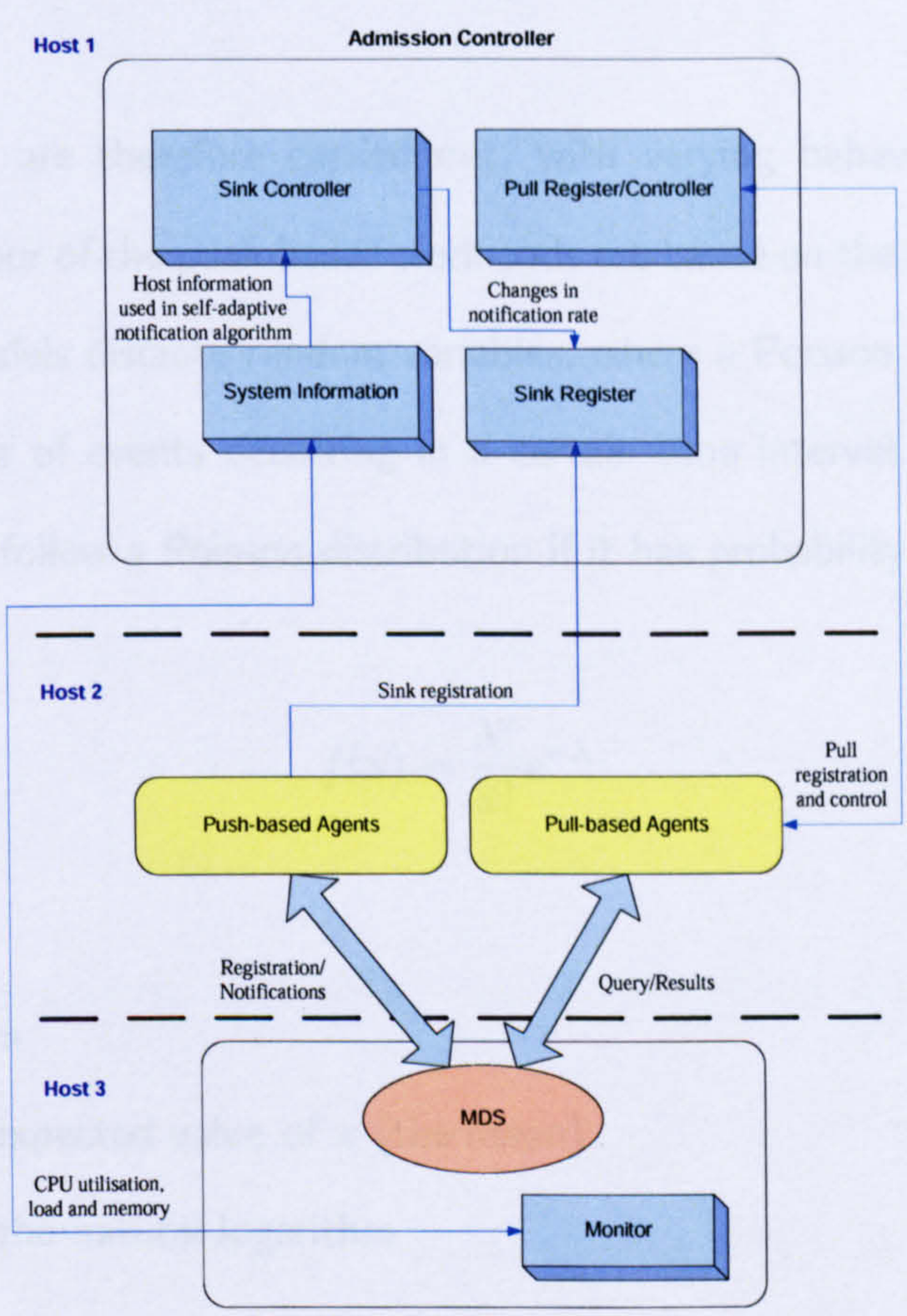

Figure 6.35: System components and their interaction in the self-adaptation for both push and pull agents with admission control.

MDS host collects system information including CPU utilisation, memory usage and load averages, for the implementation of the self-adaptive notification algorithm in the sink controller. Moreover, the pull register/controller is responsible for implementing the self-adaptive pull algorithm through multiple linear regression. The admission controller needs information from both the sink and pull controllers to block queries from the pulling agents. Additionally, the different push-based workload applied to the Index Service take place on *Host2*.

Several experiments are therefore carried out, with varying behaviour for push-based queries. The behaviour of the push-based workloads are based on the *Poisson distribution*. This distribution models discrete random variables, where a Poisson random variable is a count of the number of events occurring in a certain time interval. A discrete random variable x is said to follow a Poisson distribution if it has probability distribution

$$f(x) = \frac{\lambda^x}{x!}e^{-\lambda} \tag{6.3}$$

where

$x = 0, 1, 2, ..., n$

$\lambda > 0$, λ is the expected value of x (the mean)

e is the base of the natural logarithm

The Poisson distribution is used to modify the increasing and dynamic workloads used previously in this chapter. For the new increasing workload (*increasing Poisson*) workload, Poisson with a mean of 25.0 is used to calculate the cycle time for increasing push-based agents. However, the number of push-based agents is still increased by 18 during each cycle. Poisson with mean 25.0 is also used for the cycle time for the new dynamic workload (*dynamic Poisson*). The number of push-based agents to add or subtract is also calculated with Poisson of mean 100.0. Moreover, the decision to add or subtract agents is handled by the *RANMAR* function [75]. In the following experiments, the pull-based

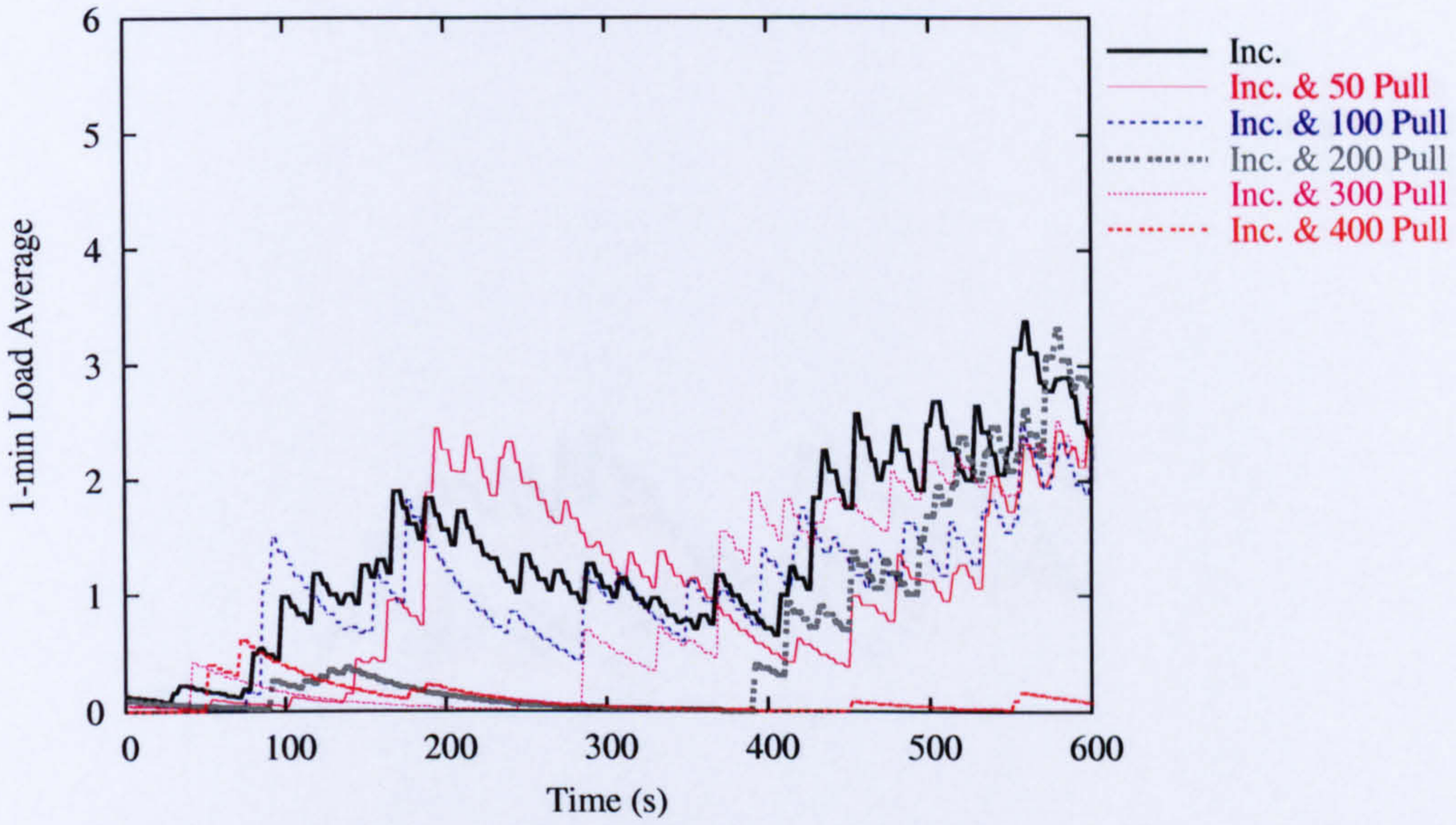

Figure 6.36: 1 min load average for the admission control of pull-based queries with push-based (increasing workload) agents.

agents are still self-adaptive and are added at the beginning of the experiments. The same self-adaptive notification algorithm is also applied to the push-based agents.

6.3.10 Admission Control with Increasing and Dynamic Workloads

The experiments in this sub-section are similar to those in Section 6.3.7, except that the admission control algorithm previously described, is also utilised. Figure 6.36 shows that all the experiment scenarios register a 1 min load average which is above zero, indicating that there are sinks being registered. This is depicted in Figure 6.48 which shows that 150 sinks are able to register when there are 400 concurrent pull-based agents. Additionally, Figure 6.37 shows that the 1 min load average is also relatively low overall; however, the load average is slightly higher than for the increasing workload, for more than around 200 pulling agents. This is explained in Figure 6.48 by the larger number of sinks than pull-based agents for more than 200 of the latter.

Figure 6.38 shows that there is more CPU activity across the various experiment scenar-

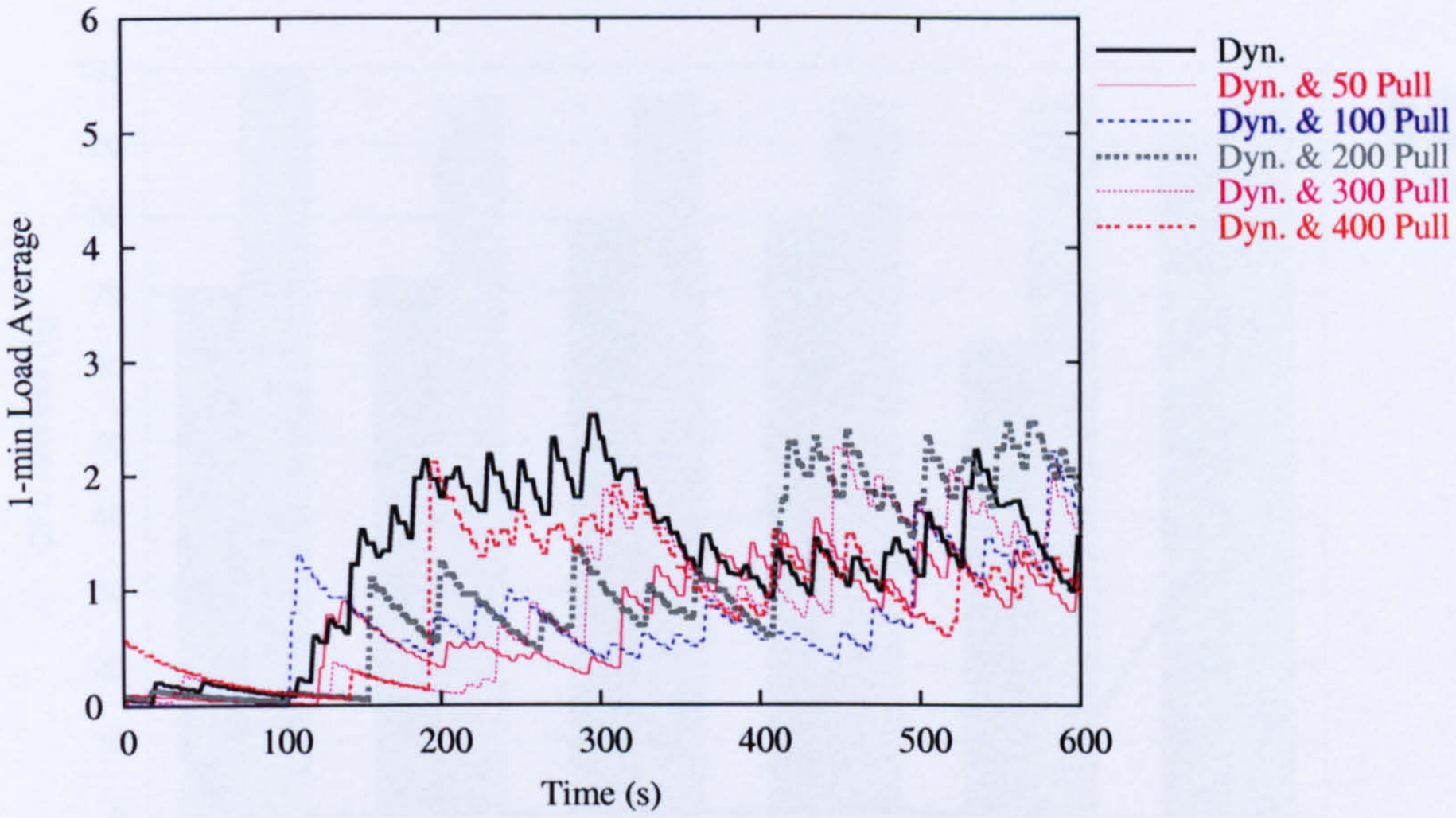

Figure 6.37: 1 min load average for the admission control of pull-based queries with push-based (dynamic workload) agents.

ios, as compared with the CPU idleness graph 6.25 (lower mean). This is because the admission control algorithm now allows more sinks to register with the Index Service. For the dynamic workload, the CPU idleness graph in Figure 6.39 is comparable with the increasing workload one, but with slightly more drops in CPU idleness.

Figures 6.40 and 6.41 both show uniform increases in the usage of memory which is noticeably higher than in Section 6.3.7. The memory usage is directly proportional to the larger number of sinks being registered via admission control.

6.3.11 Admission Control with Increasing Poisson and Dynamic Poisson Workloads

This sub-section gives the results of similar experiments as in Section 6.3.10 but with the newly defined workloads 6.3.9. Figure 6.42 shows that the 1 min load average is lower for less than 200 pulling agents, than in the increasing workload. This is due to the smaller number of sinks registered with the increasing poisson workload. The smallest number of sinks able to register in the increasing poisson workload is around 100. The reverse is

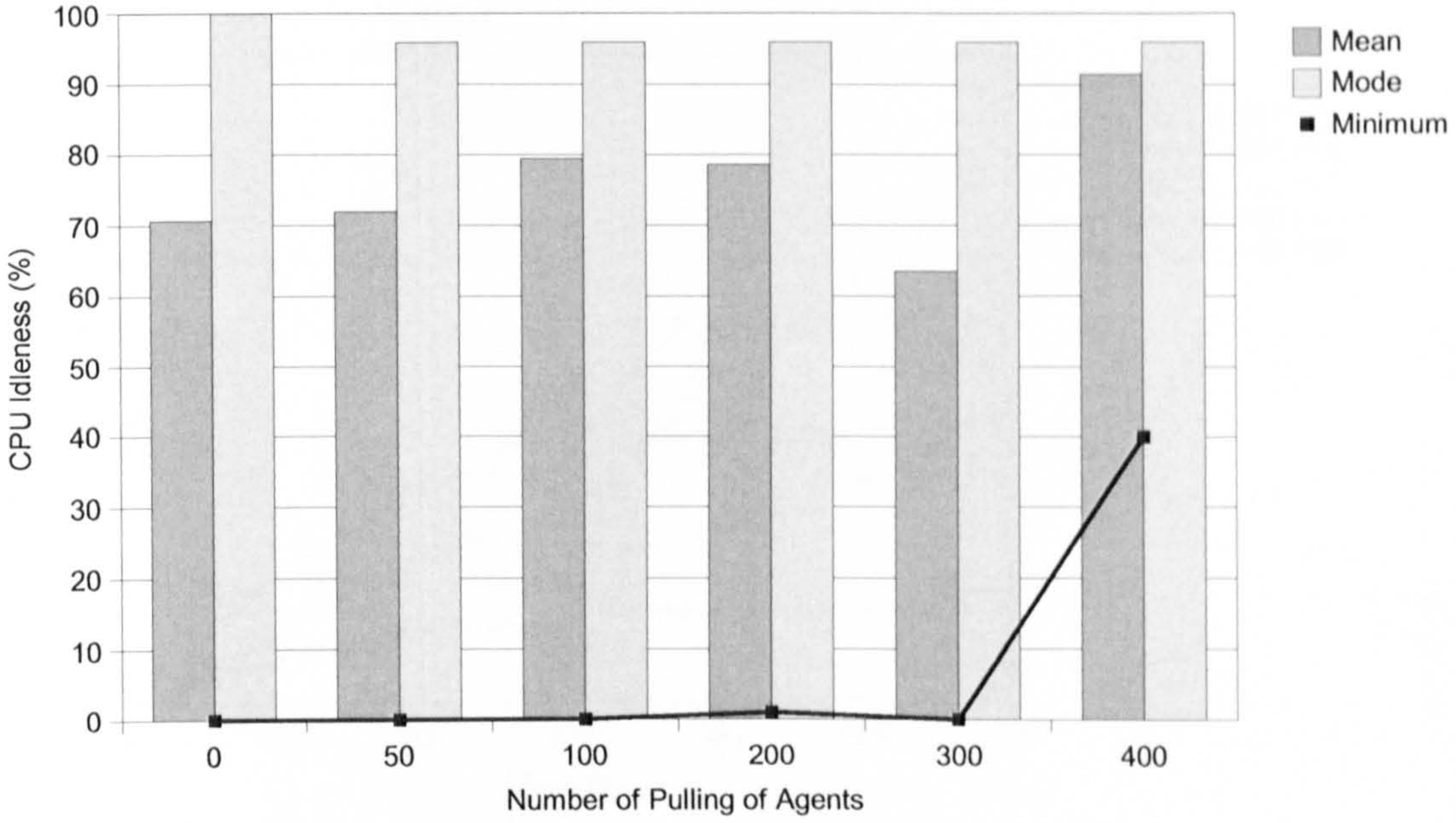

Figure 6.38: Percentage of CPU idleness for the admission control of pull-based queries with push-based (increasing workload) agents.

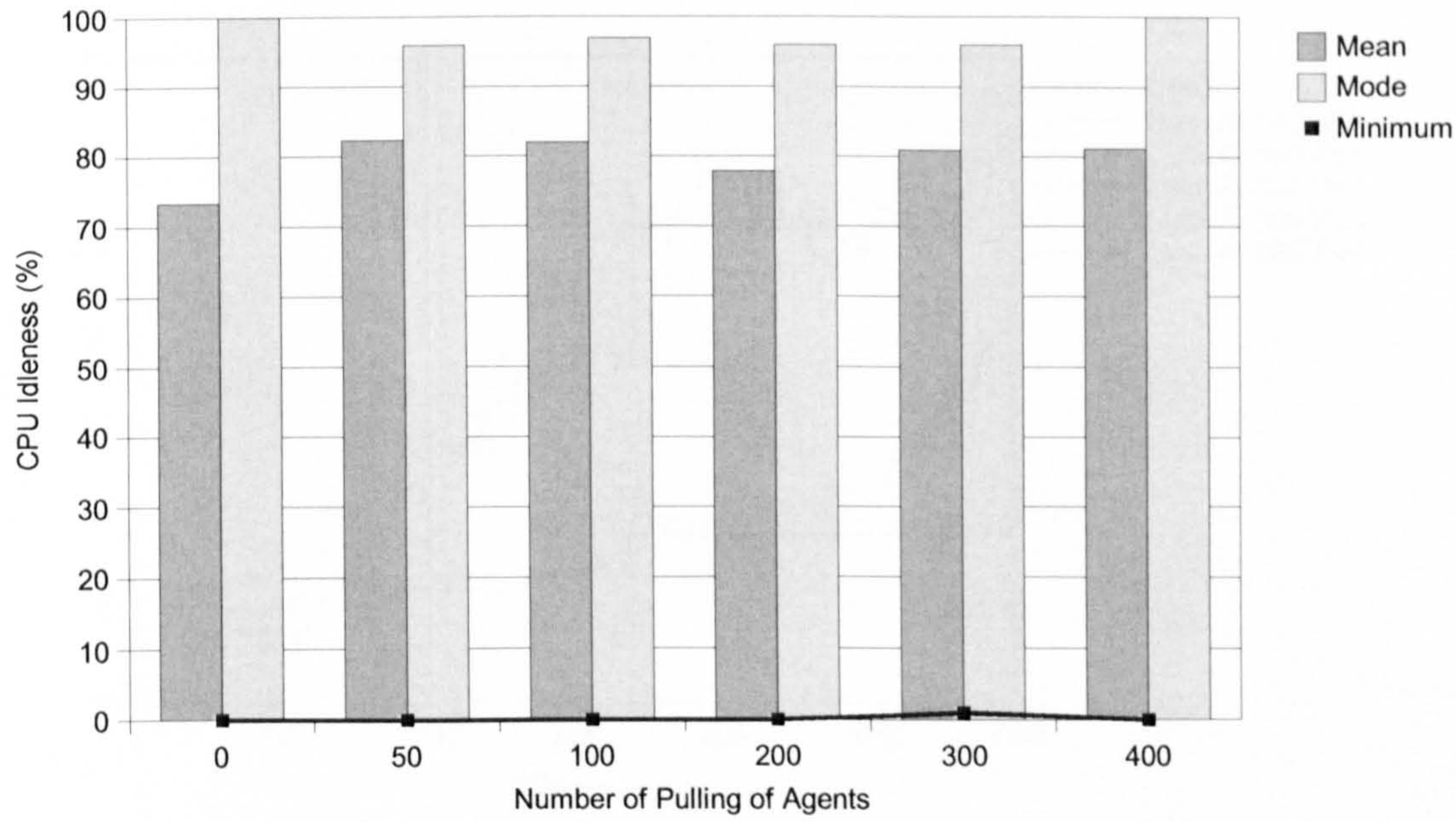

Figure 6.39: Percentage of CPU idleness for the admission control of pull-based queries with push-based (dynamic workload) agents.

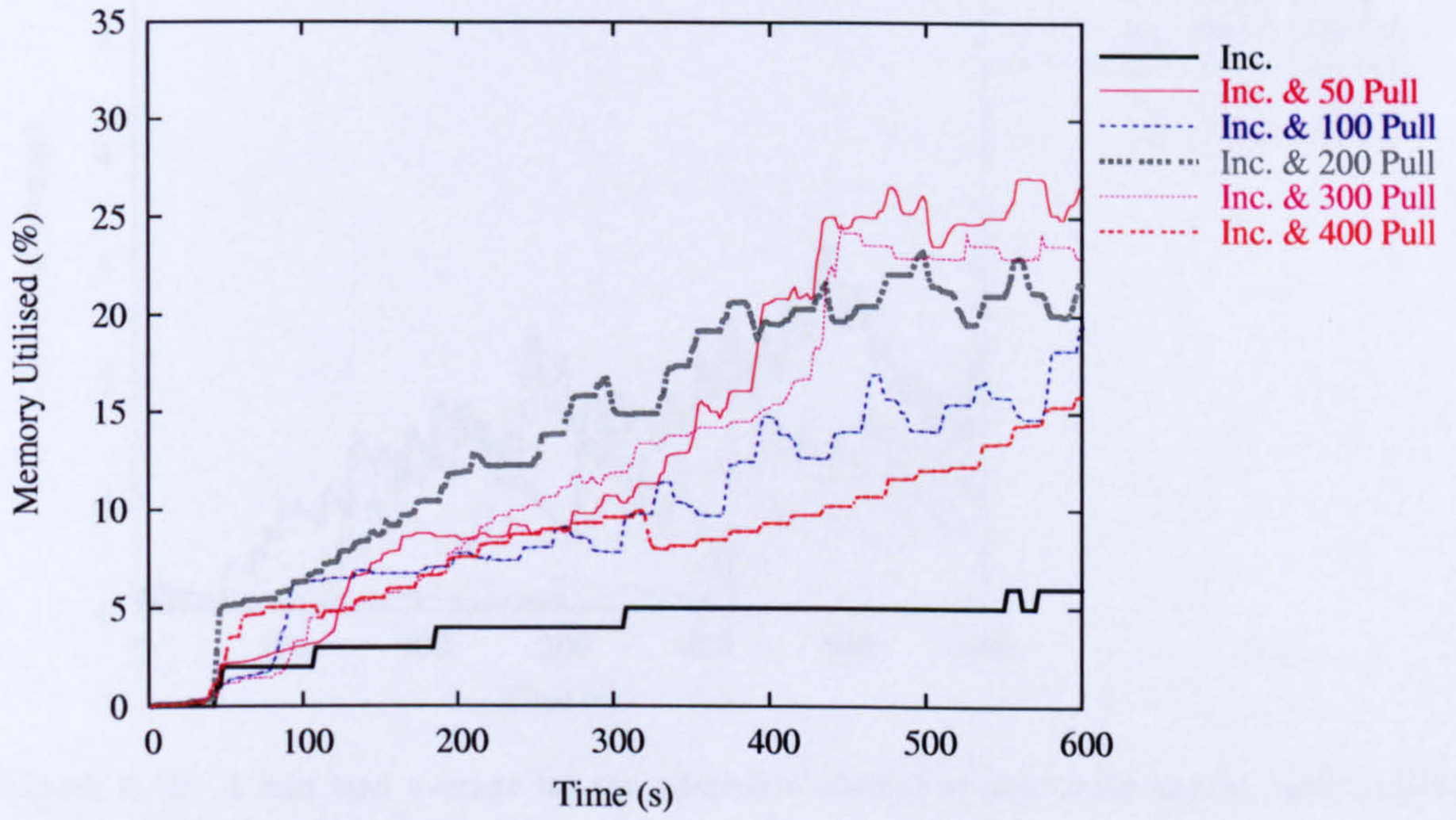

Figure 6.40: Percentage of memory utilisation for the admission control of pull-based queries with push-based (increasing workload) agents.

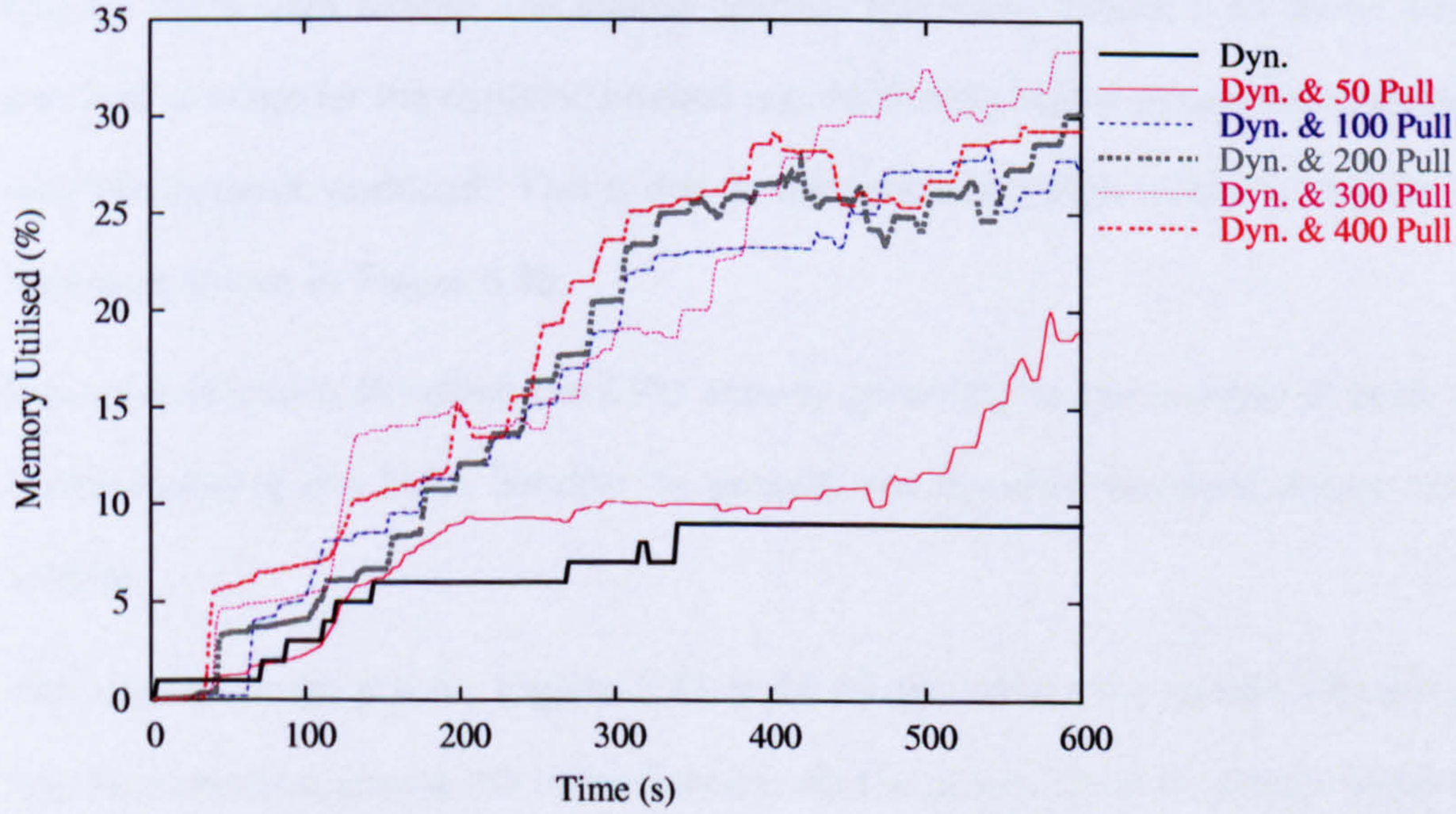

Figure 6.41: Percentage of memory utilisation for the admission control of pull-based queries with push-based (dynamic workload) agents.

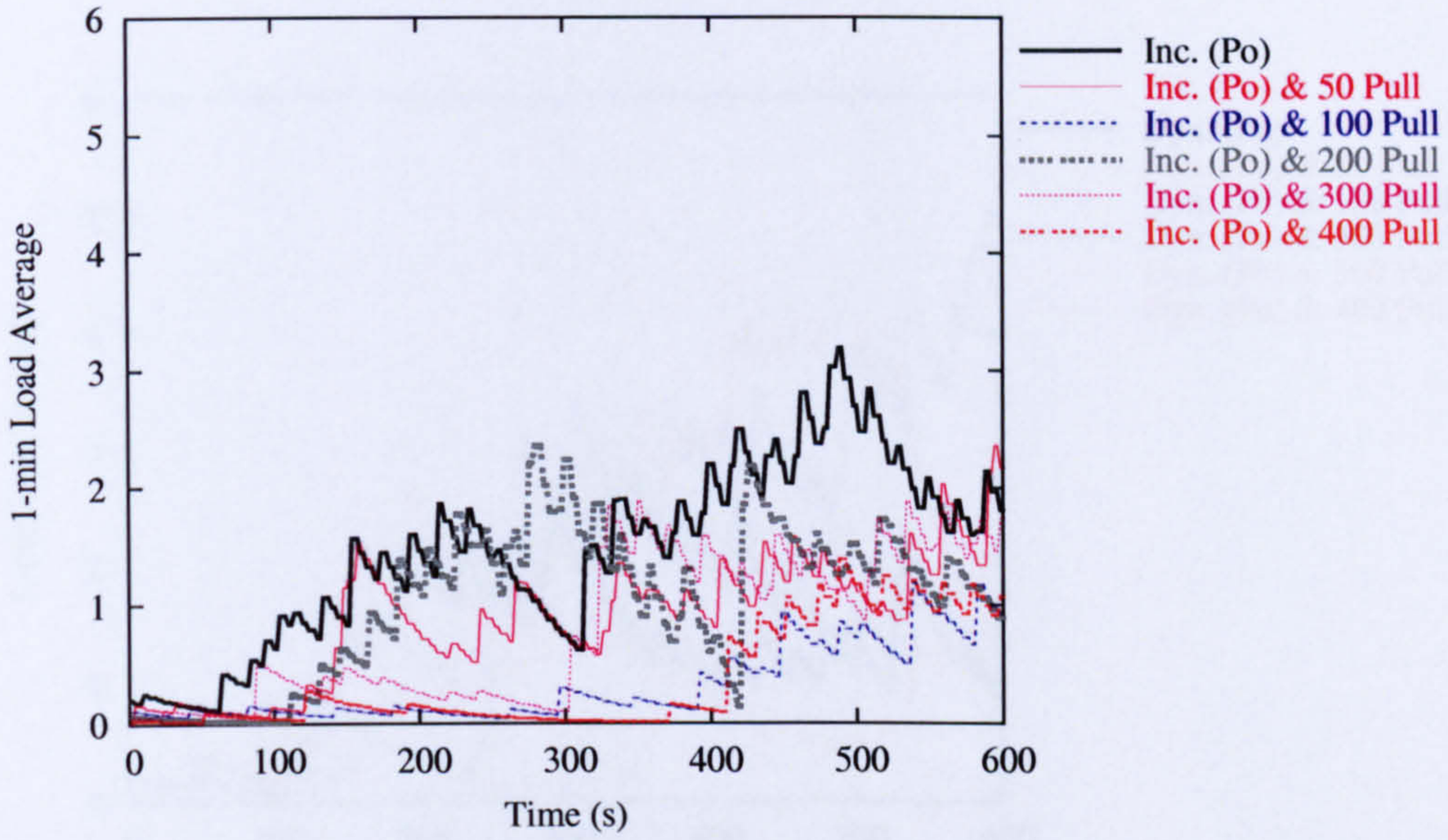

Figure 6.42: 1 min load average for the admission control of pull-based queries with push-based (increasing Poisson workload) agents.

true for more than around 250 pulling agents. Moreover, Figure 6.43 shows that the 1 min load average for the dynamic poisson is considerably higher across the scenarios, than with the dynamic workload. This is due to the concurrent high numbers of push and pull agents as shown in Figure 6.48.

Figures 6.44 and 6.45 reflect the CPU activity according to the number of push and pull agents querying the Index Service. In general, the dynamic workload shows more CPU activity.

The memory usage graphs, Figures 6.46 and 6.47 also reflect the number of push and pull agents currently querying the Index Service. All the graphs show an almost linear increase in memory usage except for the scenarios not involving any pull agents. The dynamic Poisson workload has a relatively higher memory utilisation due to the high numbers of sinks, for example 330 sinks with 400 pull agents.

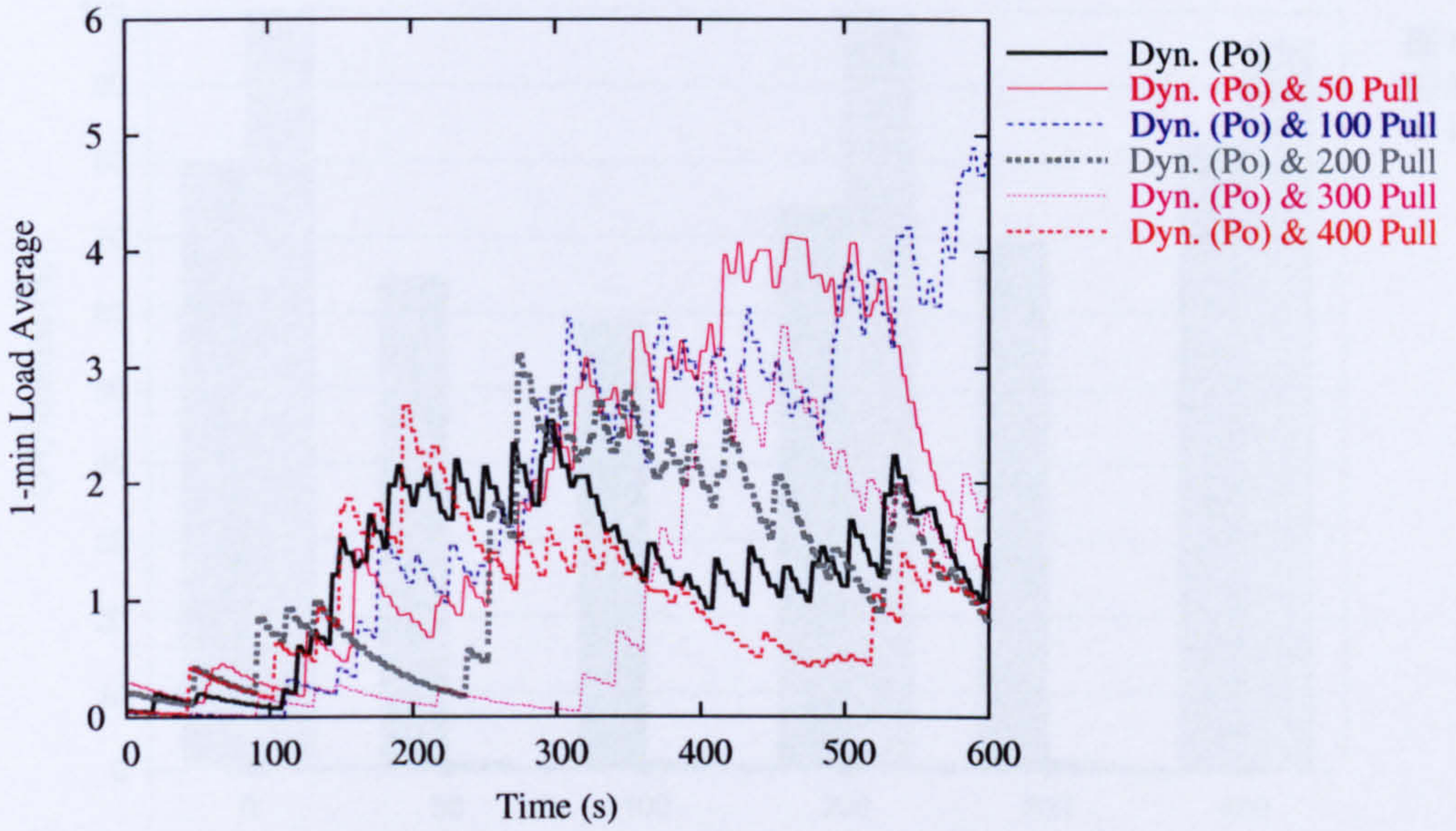

Figure 6.43: 1 min load average for the admission control of pull-based queries with push-based (dynamic Poisson workload) agents.

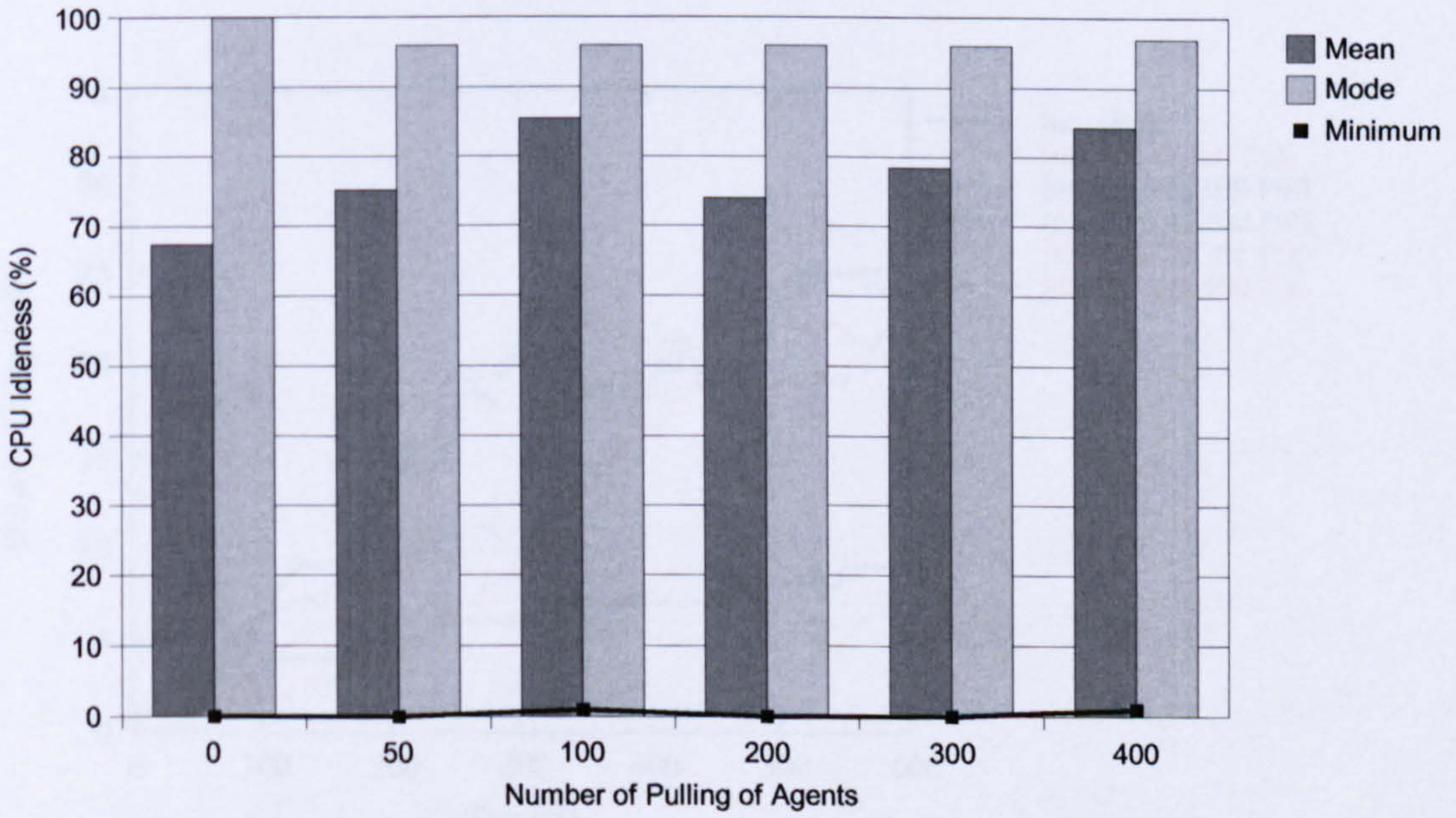

Figure 6.44: Percentage of CPU idleness for the admission control of pull-based queries with push-based (increasing Poisson workload) agents.

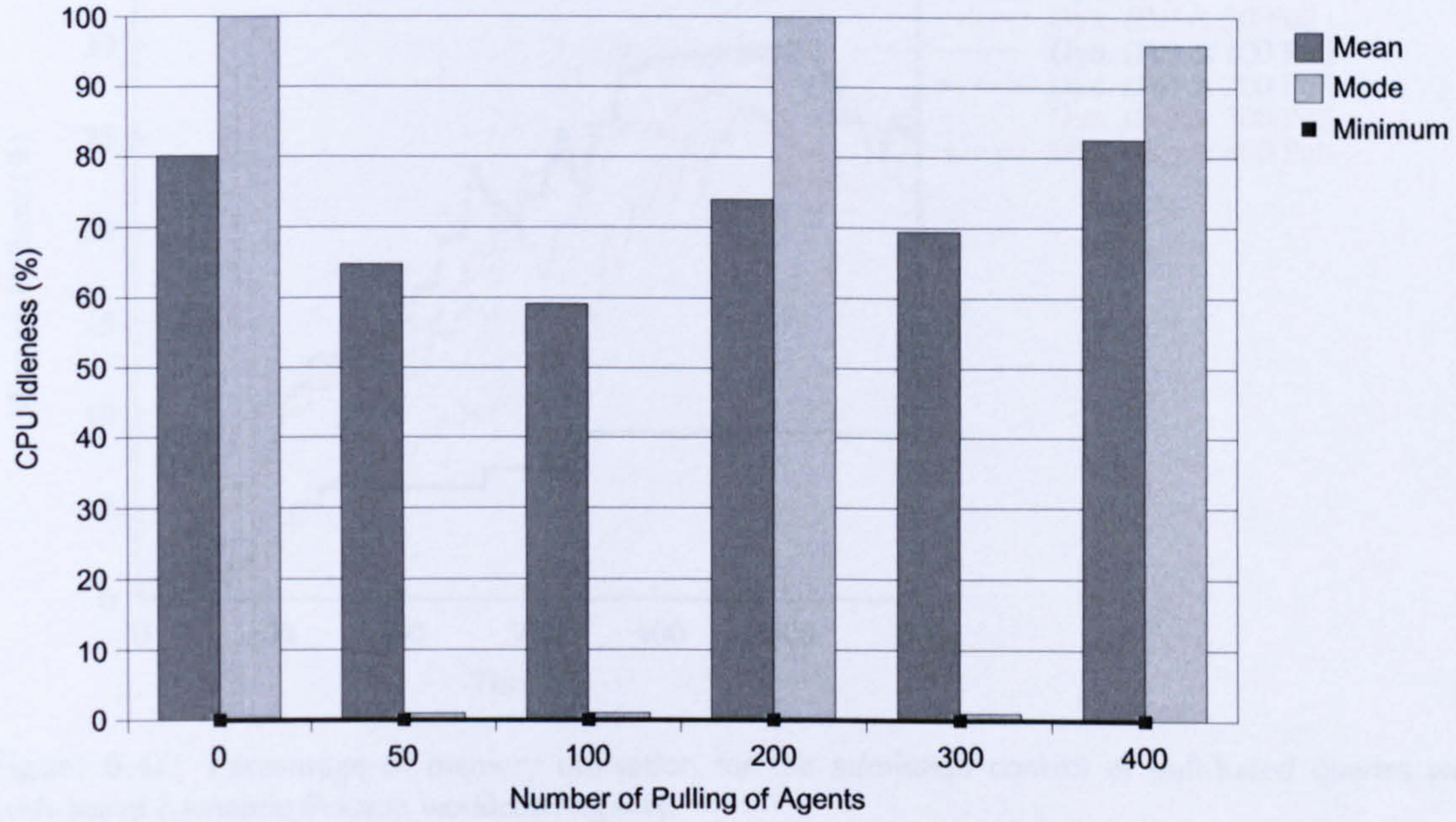

Figure 6.45: Percentage of CPU idleness for the admission control of pull-based queries with push-based (dynamic Poisson workload) agents.

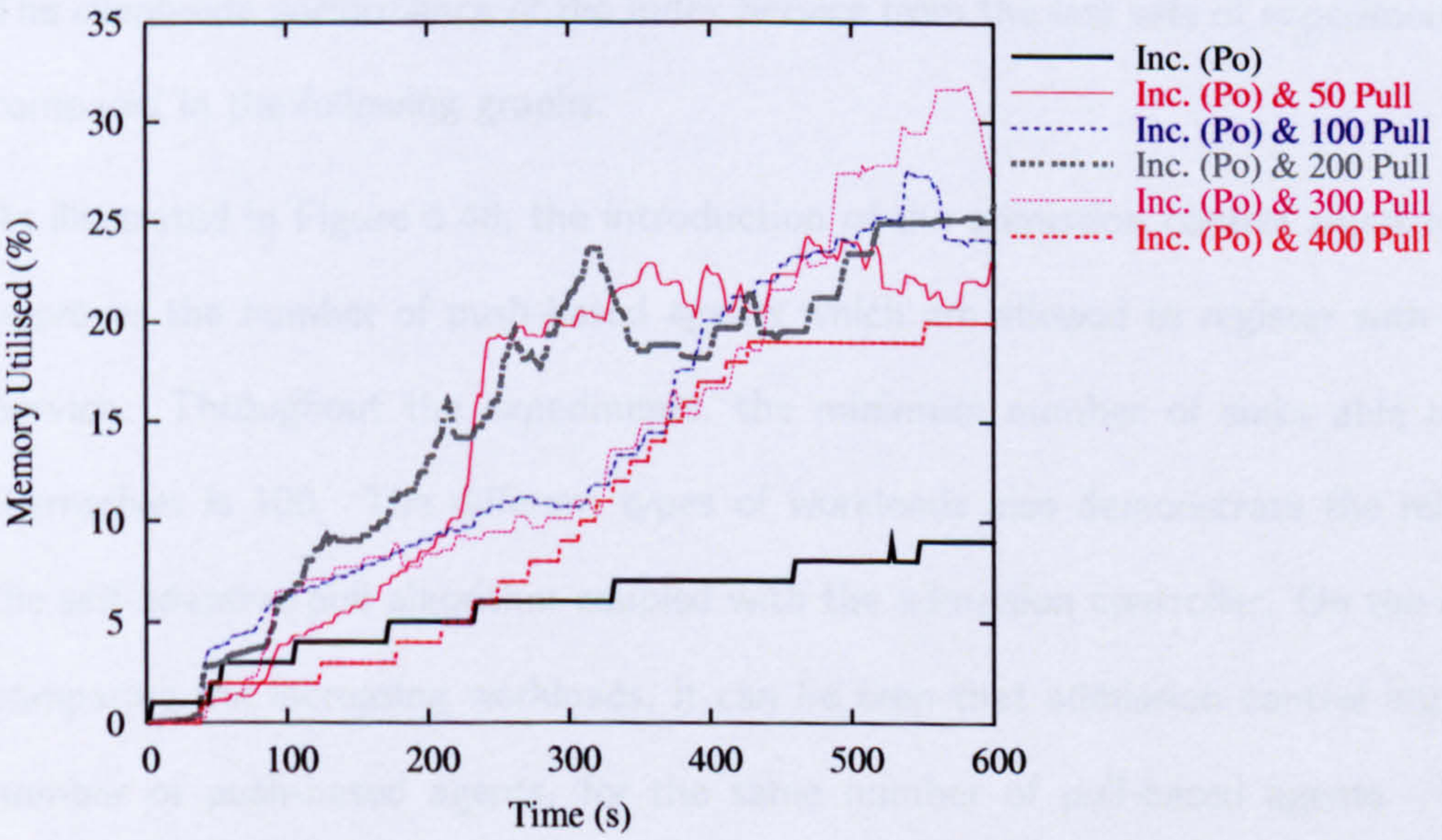

Figure 6.46: Percentage of memory utilisation for the admission control of pull-based queries with push-based (increasing Poisson workload) agents.

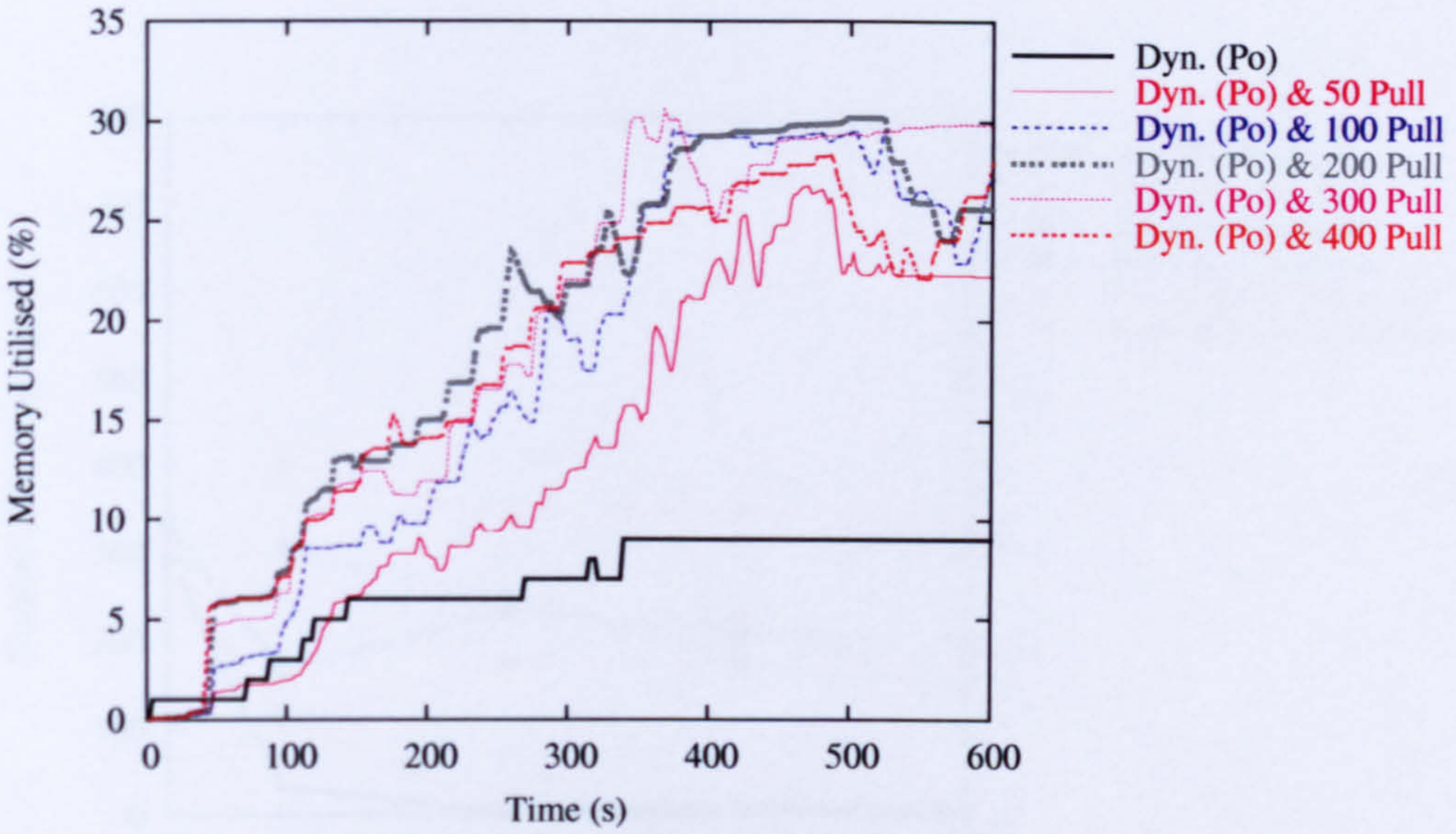

Figure 6.47: Percentage of memory utilisation for the admission control of pull-based queries with push-based (dynamic Poisson workload) agents.

6.3.12 Comparison of Client-Side Parametrics with Self-adaptive Push and Pull Queries with Admission Control

The client-side performance of the Index Service from the last sets of experiments, is now compared in the following graphs.

As illustrated in Figure 6.48, the introduction of the admission control algorithm greatly improves the number of push-based agents which are allowed to register with the Index Service. Throughout the experiments, the minimum number of sinks able to register themselves is 100. The different types of workloads also demonstrate the reliability of the self-adaptive pull algorithm coupled with the admission controller. On the one hand, comparing the increasing workloads, it can be seen that admission control improves the number of push-based agents, for the same number of pull-based agents. The same results apply to the dynamic workloads. For example, when the number of pulling agents is 400 and with increasing workloads, the minimum number of push-based agents is 150, whereas without admission control, that number tends towards zero.

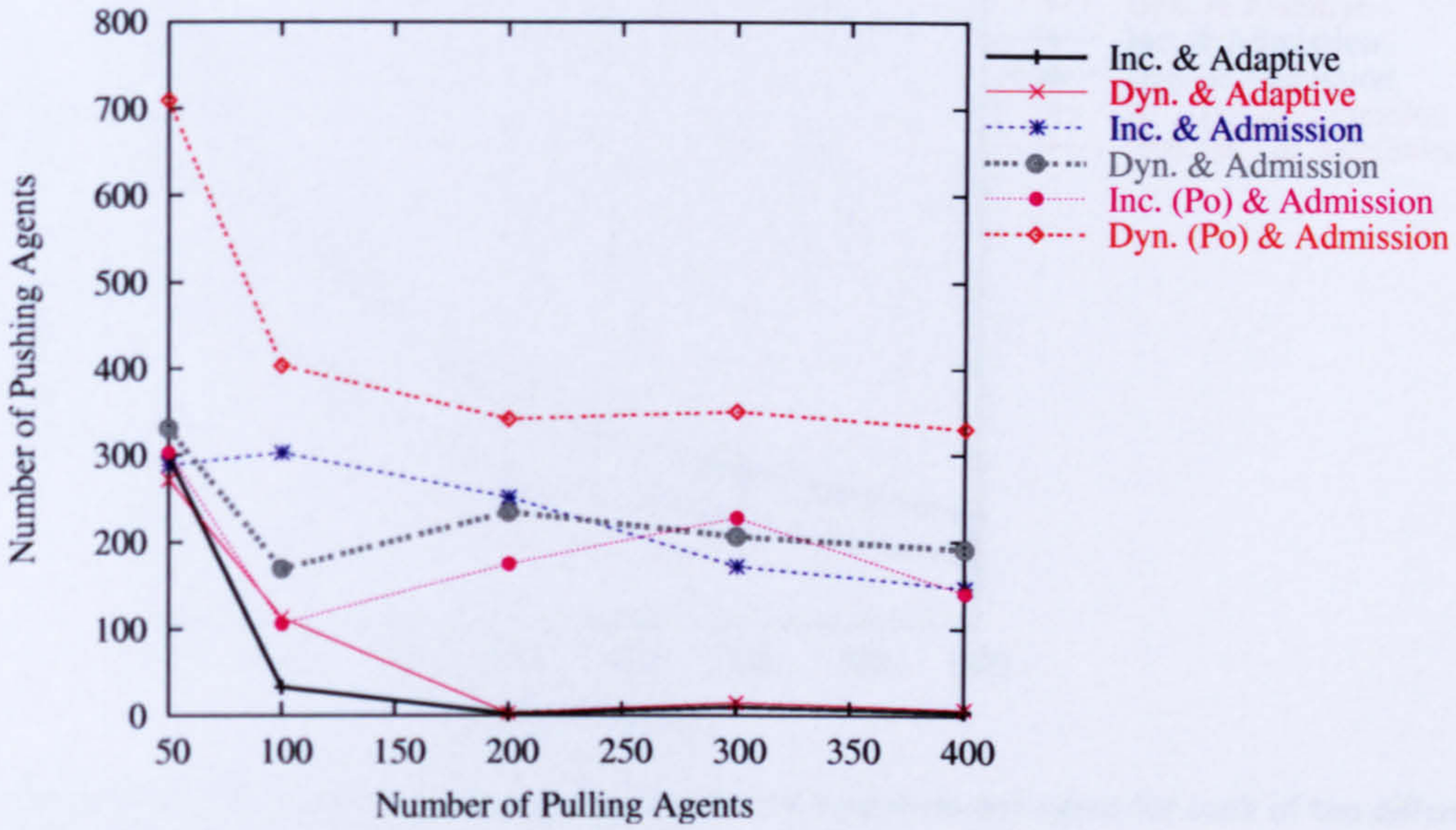

Figure 6.48: Comparison of the number of push- and pull-based agents serviced for each of the different experiments.

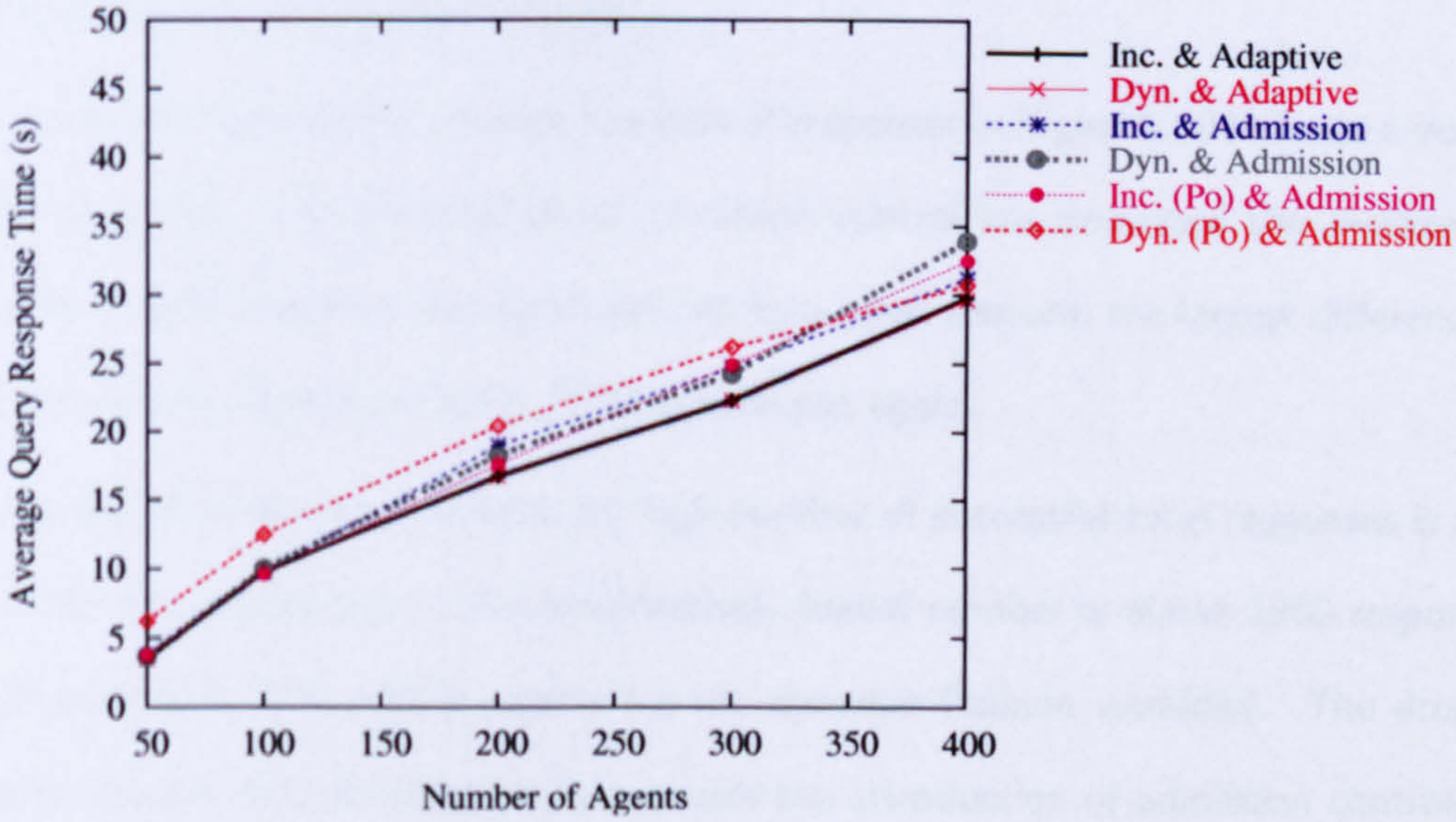

Figure 6.49: Comparison of the average query response time for each of the different experiments, against the number of pulling agents.

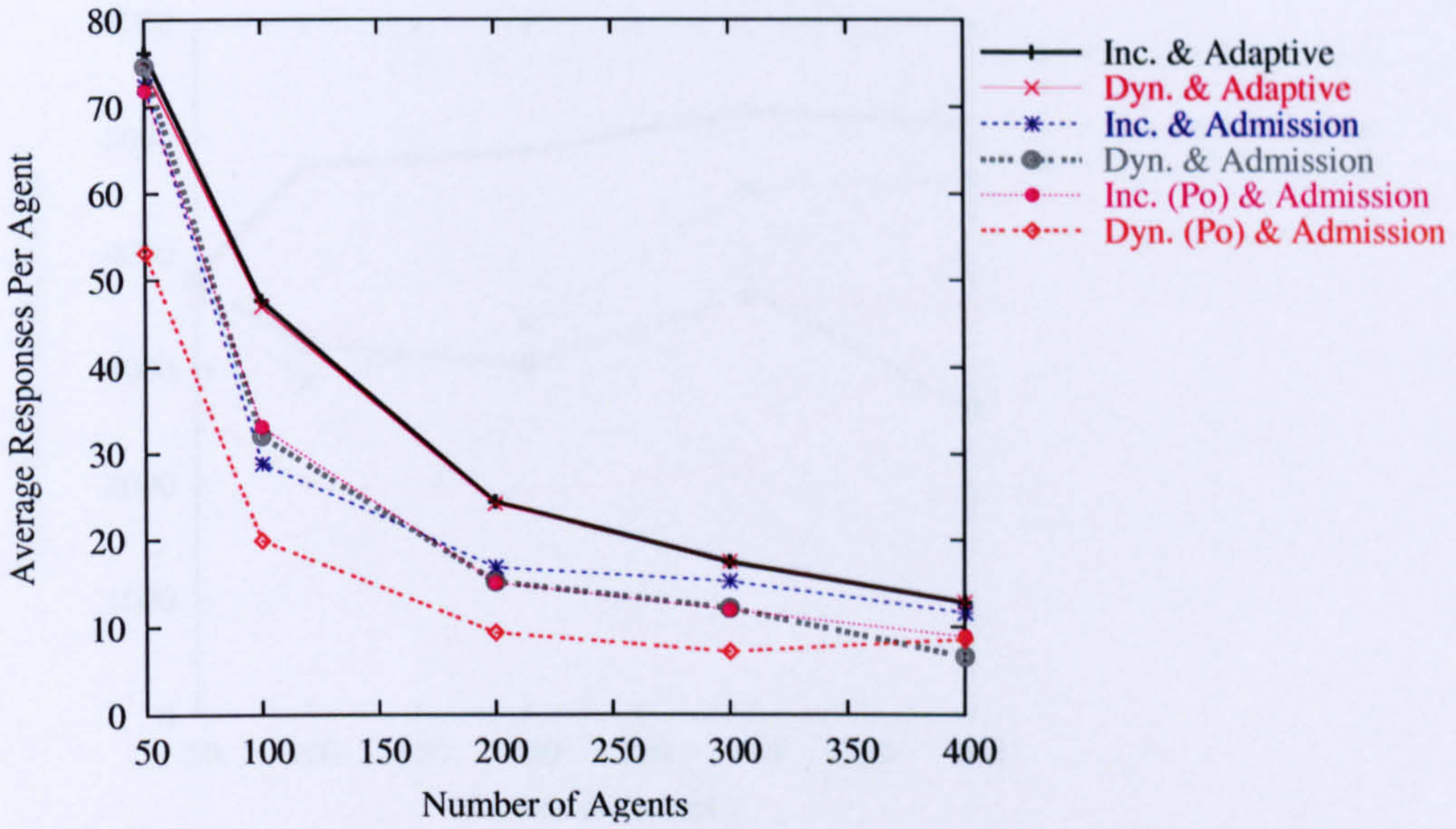

Figure 6.50: Comparison of the average number of responses per agent for each of the different experiments, against the number of pulling agents.

Figure 6.49 shows that there is a very small margin of difference in the results of the various experiment scenarios, as they all follow a similar trend. The dynamic Poisson workload has a slightly higher average response time due to the higher numbers of sinks being notified by the Index Service.

The results showing the average numbers of responses in Figure 6.50 indicate a very similar trend as well. The introduction of admission control has decreased the average number of successful responses per agent but not by a great amount; the largest difference at any point in the experiment being 18 responses per agent.

Figure 6.51 shows that a relatively high number of successful total responses is achieved across the experiments. The comparatively lowest number is about 1900 responses corresponding to 200 pulling agents for the dynamic Poisson workload. The drop in the total number of pull-based responses with the introduction of admission control, can be explained by an increase in the number of push-based sinks being able to register themselves with the Index Service. Whereas previously, that number was nearly zero, it is now in the hundreds. The counter-effect, which is deemed reasonable, is the slight decrease

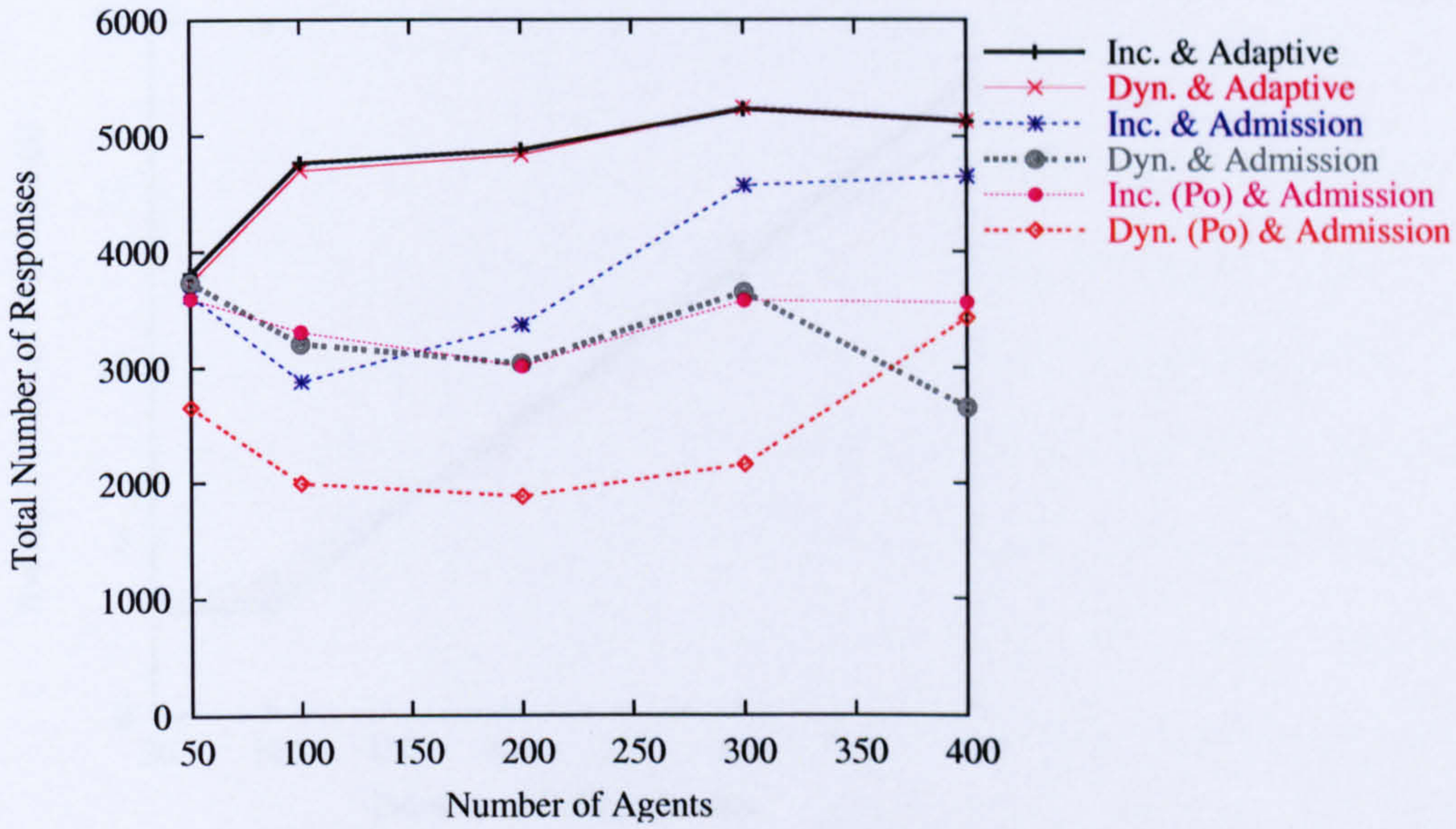

Figure 6.51: Comparison of the total number of responses for the duration of the experiment for each of the different experiments, against the number of pulling agents.

in the average number of responses per agent. This drop is on average 5 responses per agent, as shown in Figure 6.50.

Moreover, Figure 6.52 shows that there is not much difference in the calculated wait time for each of the different experiments.

6.4 Conclusion

Self-adaptive algorithms are developed in this chapter, based on the characterisation of performance benchmark data. This is carried out for notification sinks alone, and for both push- and pull-based queries. It can be seen that the self-adaptive algorithms for both push and pull queries allow a balance of sinks and synchronous queries without any detrimental effect to either. The performance of these algorithms is verified using a number of different workloads which cover both the randomness in the arrival of queries and the cumulative number of queries. Whilst notification sinks are regulated through their notification rate, depending on the number of sinks and the current CPU utilisation,

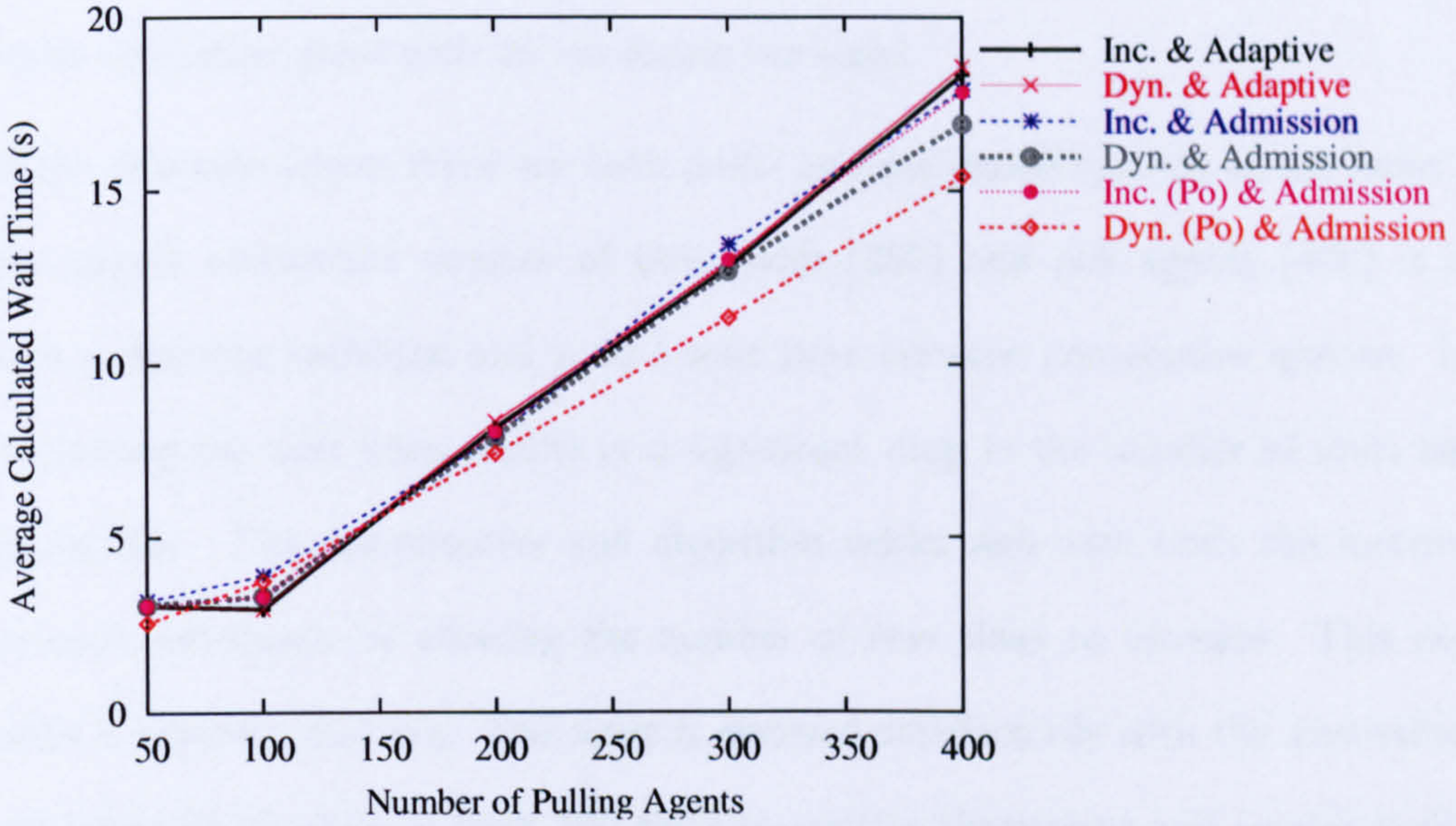

Figure 6.52: The average calculated wait time via multiple linear regression as the number of pull-based agents increases.

pull-based clients space out their queries with an optimum wait time. This produces an improvement in the performance of the pull-based clients. Subsequently, an admission control mechanism is introduced, resulting in a better balance between the numbers of each type of queries, without any loss in the client-side performance.

Whilst the Index Service can stop the registration of notification sinks, it originally has no mechanism for preventing pull queries until the latter have already been made. The self-adaptive pull algorithm and admission controller in this chapter are successful in limiting the number of pull-based queries at any one time by increasing the wait time.

The experimental results with the self-adaptive notification algorithm indicate that the sink-based algorithm is a reliable method for ensuring that the CPU utilisation of the Index Service host is kept under control. Nevertheless, this algorithm is not scalable due to the dependency on the particular Index Service host. The CPU utilisation-based notification algorithm however, takes into account the CPU utilisation available as well, thereby being more scalable. It also results in a higher notification rate than with the sink-based algorithm, which is more likely to meet users' requirements. The self-adaptive

notification algorithms proposed are useful since the load and CPU utilisation would quickly reach saturation point with an increasing workload.

In the situation where there are both push- and pull-based queries at the Index Service, the largest concurrent number of both push (350) and pull agents (400) is achieved with a dynamic workload and a 30 s wait time between consecutive queries. However, decreasing the wait time, results in a significant drop in the number of sinks being able to register. The self-adaptive pull algorithm works well with both the increasing and dynamic workloads by allowing the number of new sinks to increase. This number of sinks is however, still low. This issue is resolved satisfactorily with the admission control algorithm which allow at least 100 sinks to register themselves and receive notifications, throughout the experiments. The average response time and average number of responses per agent are also satisfactory, when tested with the dynamic, increasing and their Poisson equivalents.

Chapter 7

Conclusions

This thesis describes a self-adaptive and self-optimising system, GridAdapt, which leverages the Globus MDS3, using benchmarks for both synchronous and asynchronous query operations, to improve the performance of both the MDS and users. This chapter presents a summary of the main areas of work achieved in this thesis, as well as suggests future work.

7.1 Summary of Thesis

Chapter 2 gives details of the Grid architecture and focuses on the different Grid discovery and monitoring tools available. The work in this thesis is based on the Globus Monitoring and Discovery System, of which two versions are described in detail in Chapter 3. The OGSA-compliant MDS3 is first described, followed by its predecessor, the LDAP-based MDS2.

Being able to discover and monitor resources is crucial in the context of a distributed system, where those resources are located in multiple locations. If such capability were not in place, detailed knowledge of accessible components would not be possible. Discovery allows the identification of resources and services which have desired attributes. On the other hand, monitoring enables the continuous observation of state. Both these important functions require the ability to aggregate information from multiple, distributed

information sources.

Performance is an important issue for Grid applications running on top of middleware services. For example, the overall performance of an application will be partly dictated by the performance incurred using a resource discovery service. This performance issue is further complicated by the resources being heterogeneous and by the Grid fabric having variable bandwidth and latencies. Such a dynamic and heterogeneous environment could result in unpredictable and unreliable outcome for applications. As a result, this situation hinders the forecasted scheduling of resources and the accountability of applications.

7.1.1 Review of Thesis Contributions

- **A comparative performance study of MDS2 queries at the local levels, with different back-end implementations, and the effect on Grid applications.**

  In the first part of this thesis, detailed benchmarking studies are carried out for the characterisation of the performance of the MDS. Performance analysis is carried out at different levels in the MDS2 hierarchy, namely at the information providers and the GRIS. Various information provider implementations are also investigated and compared, as well as information update mechanisms including caching. It has been observed that the type of GRIS back-end implementation is likely to affect the performance seen by the user. A balance should also be struck between server- and client-side performance, as caching information in the GRIS greatly reduces the query average response time as expected, but the load on the GRIS host is relatively high. Other areas where attention should be given to ensure performance include installing the GIIS on the same node as the GRIS and the expected average response time which is dictated by the information provider provision mechanism.

- **The performance prediction of MDS2 for global level queries and comparison of various predictive algorithms.**

A number of predictive algorithms are also applied to the GIIS performance data which has been collected, allowing the future performance of the GIIS to be characterised qualitatively. Therefore, the behaviour of the MDS from a Grid application's point of view can be predicted fairly accurately. By using these predictions, Grid applications can decide which GIIS will show reliable performance when queries are sent to it. The 95% confidence intervals and standard deviation of the different predictors are calculated, results of which show that the AR(1) and AR(2) predictive methods are accurate. Moreover, from the confidence interval values shown, the difference in the performance of the various predictors is shown to be statistically very small.

- **The integration of the Titan scheduler with the MDS2 to allow for the publication of job information.**

  A new schema and information providers are added to the MDS to include job information from Titan. The Titan information is stored in a back-end relational database and by modifying the LDAP schemas, the distinguished names of the scheduling attributes are automatically reconfigured, thus being available for access across the MDS hierarchy.

- **The performance analysis of queries with varying complexity for the MDS2 to test its scalability with dynamic information.**

  Experiments are carried out where requests for the scheduler information, are made using queries of greatly varying complexity, and the MDS performance analysed. The effect on the MDS of querying information in different scopes or with certain criteria, is investigated. Experimental results conclude that the boolean AND operator can lead to MDS query performance degradation, with two operands and the lazy evaluation method. It is also not advised to use the NOT operator when more than 250 agents are simultaneously querying the MDS.

- **The benchmarking of the MDS3 push and pull query mechanisms under varying query loads, and the analysis of client-side performance.**

  The study of the factors affecting the performance of MDS3, scalability and stress-tests are also carried out, with both the push and pull mechanisms of query. For push-based queries, studies are performed to achieve a balance between the rate of notification of change to very dynamic data and the load at the Index Service. For pull-based queries, the effects of various cache TTL values and wait times between consecutive queries, are determined.

- **The development of push self-adaptive algorithms based on the characterisation and recommendations gathered from the benchmark data.**

- **The formulation of a pull self-adaptive algorithm and admission controller to ensure server-side stability.**

- **A novel self-adaptive Grid Information Service which gives a performance improvement with different workloads.**

  A new system called *GridAdapt* is developed, based on MDS3 to offer self-adaptation, autonomy and sustained performance at the Index Service. GridAdapt is tested with a series of workloads and results are obtained which show that GridAdapt improves the performance achieved for both the users and the MDS. Self-adaptive algorithms are developed for both asynchronous and synchronous queries, including an admission control algorithm. The latter restricts synchronous queries from overloading the MDS, thereby stopping sinks from registering. These algorithms are derived from further benchmarks which involve both push and pull users querying the Index Service in increasing numbers and under different variables including notification rate, wait time and cache TTL. Throughout the experiments, GridAdapt allows at least 100 sinks to register themselves and receive notifications, while the average response time and average number of responses per agent are also satisfactory, when tested

with the dynamic, increasing and their Poisson equivalents.

In summary, the above thesis contributions give recommendations about the various conditions which must exist to ensure the performance of both the Grid Information Service and that of users. They also highlight the numerous criteria which influence the way in which the MDS can exhibit its performance.

7.2 Future Work

Suggestions for future work include the addition of features to the admission controller component of GridAdapt, as well as the integration of trust models. Another suggestion for future work is the application of GridAdapt to MDS4 which is a WSRF implementation of information services released with Globus Toolkit 4.0 [31] in April 2005. Additionally, GridAdapt can be extended with a case study of the arrival patterns of synchronous and asynchronous queries to the MDS, which can enhance the self-adaptive algorithms through forecasting and the use of statistical methods. GridAdapt can also be integrated with GRAM which can interface local scheduling systems like Titan.

7.2.1 Admission Controller

The wait time calculated by multiple linear regression can be made to be variable across the current agents, depending on the average response time expected by the agents. Some of the pull-based clients can have a fixed wait time, while others can also take into account the expected average response time of individual agents. Alternatively, the calculated wait time can be average response time-biased, where all the pulling agents expect their query results within a certain time limit. Moreover, GridAdapt could be extended to allow each agent to switch between push and pull queries.

Furthermore, it would be useful for GridAdapt to block certain synchronous queries al-

together, to prevent the Index Service from overloading. The corresponding problem is for GridAdapt to be aware which users are likely to query the Index Service repeatedly in short bursts. A trust model is therefore being developed based on previous work [26], to map the query behaviour of agents for the informed decisions of GridAdapt and early results have been promising.

The effect of various parameters on the performance of GridAdapt, can also be studied. For example, the cycle period used for the access controller to evaluate the current conditions and make a decision, can also be varied and experiments carried out to find its optimum value. Similarly, the size of the moving window for the CPU utilisation-based self-adaptive notification algorithm, can be varied and the effect on the performance optimisation of GridAdapt analysed. The moving window can then be used to calculate dynamically an accurate average value for the last CPU utilisation readings.

Performance guarantees can also be added to the Index Service to ensure a minimum notification rate for different categories of clients, each with a certain QoS. A certain level of service expectation can be attached to each notification sink, leading the Index Service to differentiate amongst these push-based clients, in terms of the performance delivered. The Index Service must therefore implement *QoS agreements* or *contracts* to achieve both a reliable level of performance and notification prioritisation; this can be carried out using *Service Level Agreements*. Subsequently, when the Index Service is in danger of overloading, the notification rate of lower priority clients, is decreased first. The overall capacity of the MDS can thus be shared across the different types of clients, so that higher priority clients do not have to experience a reduction in perceived performance.

Similar QoS notions apply to the pull-based agents but with regards to the average query response time. This type of *service distinction* will also avoid lower priority pull-based clients being made to wait for extensive periods of time between queries.

7.2.2 GridAdapt and the Grid

The research carried with GridAdapt can also be mapped to a Wide Area Network, where the effects of network latency, low bandwidth and the heterogeneity of resources can be investigated. It is envisaged that GridAdapt will work in a similar way as in this thesis, but with the added costs of networking. Various instances of GridAdapt, as shown in Figure 6.35, can easily run on different physical hosts, where each GridAdapt instance corresponds to one Index Service. In order to keep the load on the Index Service host at a minimum, the GridAdapt instance should run on a different host to which clients should issue queries. This set-up will ensure that the MDS runs on a dedicated server. Experimental studies can also be carried out to analyse the difference between the GridAdapt instance running on a the same LAN as the MDS, and in a different administrative domain.

Future work can also involve GridAdapt instances being able to self-adapt amongst themselves, by redirecting queries to other trusted instances, when the client's QoS agreement will not be met. Performance evaluation of GridAdapt can subsequently be carried out on large Grid infrastructures in realistic scenarios, for example, by using the White Rose Grid [122] or the National Grid Service [84].

7.2.3 GridAdapt and MDS4

The core framework of GT4 leverages *Web services mechanisms* for the definition of its interfaces and for the architecture of its components. These mechanisms are used for the development of service-oriented architectures and applications, where service interfaces are described uniformly.

As well as providing query and subscription interfaces to resource data, MDS4 [99] also offers a *trigger interface* which can be configured to take particular actions when pre-specified conditions are breached. MDS4 also includes an aggregator framework onto

which the WS MDS services are built. Additionally, GT4 provides a selection of browser-based interfaces, command-line tools and Web services interfaces which enable users to access and query the collected information. For instance, GT4 offers the *WebMDS* service which can be configured via XSLT transformations [15] to create specialised views of the Index Service data.

The framework and fundamental characteristics of MDS3 and those of MDS4 are similar in that they are both based on Web services, although the architecture has changed from OGSI in GT3 to WSRF in GT4. In OGSI, services advertise their *service data* while in WSRF, services make their *resource properties* available. Service data and resource properties are essentially very similar, where they both provide a mechanism for representing data in XML format. Consequently, it is envisaged that the migration of GridAdapt to MDS4 would require minimal changes. Some modifications would include service interface names; for instance, query operations would use `GetResourceProperty` and `GetMultipleResourceProperties` instead of `findServiceData`. For subscription and notification operations, the OGSI `NotificationSource` and `NotificationSink` are replaced by `WS-Notification`. Furthermore, notification and pull benchmarks can be compiled for different platforms, as part of the knowledge gathering process for GridAdapt. For example, the *MDS-Archiver* service can be used to store information source values as historical information, in a persistent database for later query. Moreover, whilst the configuration and implementation of MDS4 have been improved and more closely integrated with other components of GT4, MDS4 presents a lower performance over previous versions due to the use of XML protocols. Subsequently, GridAdapt can be used to ensure that the MDS can still perform adequately and reliably.

7.2.4 GridAdapt and Forecasting

Further techniques can be developed to predict the MDS host behaviour in the context of *computational economies* [125]. *Commodities markets* are a category of computational

economies where a purchaser of a service (for example, a querying agent) is one of many buying an entity (successful up-to-date query operations). If the query behavioural patterns (type of data queried and the frequency) of these purchasers are monitored and logged for weeks, it can be possible to analyse the data and by using time series modelling, obtain forecasts. For example, it would be useful to gather query behavioural data from production Grid. Such forecasts can then be used to enhance the performance of GridAdapt which will be able to allocate its capacity, with priority given to higher-paying users.

7.2.5 GridAdapt and GRAM

As the MDS is used to discover and monitor any type of useful information, GridAdapt could also be used to steer the querying of job execution information provided by the GRAM service. GRAM provides a uniform interface for requesting the utilisation of remote resources for the execution of jobs. It is also used for remote job control, interfacing scheduling systems. The Web Services GRAM component in GT4 consists of a set of WSRF-compliant Web services to submit, monitor and cancel jobs on distributed resources. It is also possible for a client to use GRAM to specify the type and number of resources required, the data to be staged to and from the execution target resources, and the executable and its associated arguments.

The GRAM service in GT4 defines appropriate resource properties that contribute towards service discovery and monitoring. GridAdapt could therefore allow the controlled monitoring of the status of both the computational resources and individual tasks via query operations or notification subscription. This information can run in the Index Service container as WS-Resources, and shared across locations through distributed hierarchical registries.

Appendix A

Papers Published During this Work

The following lists details of papers which were published in scientific journals, conferences and workshops during the course of this thesis.

- S. A. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. Cao, S. Saini, G. R. Nudd; 'Performance Prediction and its Use in Parallel and Distributed Computing Systems'; *Future Generation Computer Systems, Special Issue on System Performance Analysis and Evaluation*, 22(7): 745-754, 2006.

- H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, G. R. Nudd; 'Performance Evaluation of a Grid Resource Monitoring and Discovery Service'; *IEE Proc.-Software*, 150(4):243-251, August 2003.

- J. R. D. Dyson, N. E. Griffiths, H. N. Lim Choi Keung , S. A. Jarvis, G. R. Nudd; 'Trusting Agents for Grid Computing'; *IEEE International Conference on Systems, Man and Cybernetics (SMC 2004)*, The Hague, The Netherlands, October 10-13, 2004.

- H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, G. R. Nudd; 'Performance Modelling of a Self-adaptive and Self-optimising Resource Monitoring System for Dynamic Grid Environments'; *UK e-Science All Hands Conference: e-Science Broadening the Horizon (AHM 2004)*, Nottingham, August 31-September 3, 2004.

- H. N. Lim Choi Keung, L. Wang, D. P. Spooner, S. A. Jarvis, W. Jie, G. R. Nudd; 'Grid Resource Management Information Services for Scientific Computing'; *International Conference on Scientific & Engineering Computation (IC-SEC 2002)*, Singapore, December 3-5, 2002, pp.746-750.

- H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, G. R. Nudd; 'Self-adaptive and Self-optimising Resource Monitoring for Dynamic Grid Environments'; *2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS 2004), in conjunction with the 15th International Conference on Database and Expert Systems Applications (DEXA 2004)*, Zaragoza, Spain, August 30-September 4, 2004, pp.689-693.

- H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, G. R. Nudd; 'Predicting the Performance of Globus Monitoring and Discovery Service (MDS-2) Queries'; *4th ACM/IEEE International Workshop on Grid Computing (Grid 2003), held as part of SuperComputing 2003*, Phoenix, Arizona, November 17 2003, pp.176-183.

- H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, G. R. Nudd; 'The Globus Monitoring and Discovery Service (MDS-2): a Performance Analysis'; *19th Annual UK Performance Engineering Workshop (UKPEW 2003)*, University of Warwick, UK, July 9-10, 2003, pp.103-116.

- S. A. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. R. D. Dyson, L. Zhao, G. R. Nudd; 'Performance-based Middleware Services for Grid Computing'; *5th International Workshop on Active Middleware Services (AMS 2003), held as part of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12/GGF-8)*, Seattle, USA, June 25 2003, pages 151159, ISBN 0-7695-1983-0.

- S. A. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. Cao, S. Saini, G. R. Nudd; 'Performance Prediction and its Use in Parallel and Distributed Computing Sys-

tems'; *IEEE/ACM International Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems, held as part of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 22-26, 2003, page 276, ISBN 0-7695-1926-1.

- H. N. Lim Choi Keung, J. Cao, D. P. Spooner, S. A. Jarvis, and G. R. Nudd; 'Grid Information Services using Software Agents'; *18th Annual UK Performance Engineering Workshop (UKPEW 2002)*, University of Glasgow, UK, July 10-11, 2002, pp.187-198.

- D. P. Spooner, J. Cao, J. D. Turner, H. N. Lim Choi Keung, S. A. Jarvis, and G. R. Nudd; 'Localised Workload Management using Performance Prediction and QoS Contracts'; *18th Annual UK Performance Engineering Workshop (UKPEW 2002)*, University of Glasgow, UK, July 10-11, 2002, pp.69-80.

Appendix B

List of Core MDS2 Information Providers

grid-info-mds-core

Generates an `MdsSoftwareDeployment` object containing information about automatically detected data such as `$GLOBUS_LOCATION`, base DNs of servers, and `slapd` process ids; and user-configured data such as administrator's email address and comments on the deployment.

grid-info-platform-*

Generates base `MdsComputePlatform` type object, reporting the compute platform type and instruction-set architecture. `grid-info-platform-uname` works generically on most Unix variants.

grid-info-os-*

Generates an `MdsSoftwareDeployment` object containing information about the host operating system, as well as extension `MdsOperatingSystemSummaryInfo` data for the base platform object. `grid-info-os-uname` works generically on most Unix variants.

grid-info-cpu-*

Generates `MdsCpuSummaryInfo`, `MdsDeviceClassCpu`, `MdsDeviceCpu` objects describing the CPUs available on the system.

grid-info-cpu-linux interprets Linux 2.2/2.4 "`/proc/cpuinfo`".

grid-info-cpu-irix interprets Irix 6.4/6.5 "`hinv`" program output.

grid-info-cpu-solaris interprets Solaris "`sysinfo`" program output.

grid-info-cpu-aix interprets AIX "lsattr" and "`lsconfig`" program output.

grid-info-cpu-tru64 interprets Tru64 "helper executable" program output.

grid-info-cpufast-*

Rapidly generates same output of `grid-info-cpu-*` by reading an inventory output from a cache file and updating only the "CPU free" attributes using system load-average information. This provides high-frequency load information at reduced cost.

grid-info-cpufast-uptime interprets "`uptime`" program output.

grid-info-mem-*

Generates `MdsPhysicalMemorySummaryInfo`, `MdsVirtualMemorySummaryInfo`, `MdsPhysicalMemory`, and `MdsVirtualMemory` objects describing host RAM and VM status.

grid-info-mem-linux interprets Linux 2.2/2.4 "`/proc/meminfo`".

grid-info-mem-irix interprets Irix "`top`" program output.

grid-info-mem-solaris interprets Solaris "`top`" program output.

grid-info-mem-aix interprets AIX "`lsps`", "`lscfg`", and "`lsattr`" program output.

grid-info-mem-tru64 interprets Tru64 "`vmstat`" program output.

grid-info-net-*

Generates `MdsNetworkInterfaceSummaryInfo`, `MdsDeviceClassNetworkInterface`, and `MdsDeviceNetworkInterface` objects.

grid-info-net-linux interprets Linux "`ifconfig`" output for active interfaces.

grid-info-net-tru64 interprets Tru64 "`netstat -in`" numeric interface info, but does

not provide netmask info.

grid-info-net-netstat interprets Irix, Solaris or AIX "`netstat -in`" numeric interface info, but does not provide netmask info.

grid-info-fs-* `[-scratch <mount point>]...`

Generates `MdsFilesystemSummaryInfo`, `MdsDeviceClassFilesystem`, and `MdsFilesystemInfo` objects. By default reports all local filesystems, while `-scratch <mount>` options select reporting only of local filesystem(s) at specified mount points.

grid-info-fs-posix interprets Posix "`df -l -k [<mount>]...`" output.

grid-info-fs-irix interprets Irix "`df -l -k [<mount>]...`" output.

grid-info-fs-aix interprets AIX "`df -P -k [<mount>]...`" output.

grid-info-fs-tru64 interprets Tru64 "`df -l -k [<mount>]...`" output.

Appendix C

MDS2 Configuration Files

Below are example details of the configuration files of an MDS2 installation, where both the GRIS and GIIS are installed on the host named `frog.dcs.warwick.ac.uk`.

grid-info.conf

```
########################################################################
#
# File: grid-info.conf
#
# Purpose: This file contains the configuration information
#          for the local MDS service
#
########################################################################


# These values are modifiable by the administrator

GRID_INFO_HOST="frog.dcs.warwick.ac.uk"

GRID_INFO_PORT="2135"

GRID_INFO_BASEDN="Mds-Vo-name=local, o=Grid"

GRID_INFO_ORGANIZATION_DN="Mds-Vo-name=frog, o=Grid"

GRID_INFO_ORGANIZATION_ADMIN_DN=""

GRID_INFO_TIMEOUT="30"


# Specify the administrator's e-mail address here
```

```
GRID_INFO_ADMINISTRATOR="name@dcs.warwick.ac.uk"

export GRID_INFO_HOST
export GRID_INFO_PORT
export GRID_INFO_TIMEOUT
export GRID_INFO_ORGANIZATION_DN
export GRID_INFO_ORGANIZATION_ADMIN_DN
export GRID_INFO_ADMINISTRATOR

# These values are used by several scripts

hostname="frog.dcs.warwick.ac.uk"
```

grid-info-resource-ldif.conf

```
# this file contains the default GRIS providers and must be configured
# for a particular platform to specialize the template...

# generate top-level Mds-Host-hn=host object every minute
dn: Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-platform-merged
args: -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid -validto-secs 60 -keepto-secs 60
cachetime: 60
timelimit: 20
sizelimit: 1

# generate CPU availablity information every minute
dn: Mds-Device-Group-name=processors, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
```

```
type: exec
path: /home/globus/gt2//libexec
base: grid-info-cpufast-uptime
args: -devclassobj -devobjs -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid
  -validto-secs 60 -keepto-secs 60
cachetime: 60
timelimit: 20
sizelimit: 100

# generate CPU inventory (hidden cache) every 12 hours
dn: Mds-Device-Group-name=processors, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-cpu-linux
args: -noobjs
cachetime: 43200
timelimit: 20
sizelimit: 1

# generate memory info every minute
dn: Mds-Device-Group-name=memory, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-mem-linux
args: -devclassobj -devobjs -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid
      -validto-secs 60 -keepto-secs 60
cachetime: 60
timelimit: 10
sizelimit: 3

# generate disk info every 15 minutes
```

```
dn: Mds-Device-Group-name=filesystems, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-fs-posix
args: -devclassobj -devobjs -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid
      -validto-secs 900 -keepto-secs 900
cachetime: 900
timelimit: 20
sizelimit: 20

# generate network info every 15 minutes
dn: Mds-Device-Group-name=networks, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-net-linux
args: -devclassobj -devobjs -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid
      -validto-secs 900 -keepto-secs 900
cachetime: 900
timelimit: 20
sizelimit: 20

# generate OS info every 12 hours
dn: Mds-Software-deployment=operating system, Mds-Host-hn=frog, Mds-Vo-name=local,
    o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-os-uname
args: -devclassobj -devobjs -dn Mds-Host-hn=frog,Mds-Vo-name=local,o=grid
```

```
      -validto-secs 900 -keepto-secs 900
cachetime: 43200
timelimit: 20
sizelimit: 1

# generate GRIS info every 12 hours
dn: Mds-Software-deployment=MDS GRIS, Mds-Host-hn=frog, Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: grid-info-mds-core
args:
cachetime: 43200
timelimit: 20
sizelimit: 1

# The following lines for fork entry added by setup-globus-gram-reporter
# generate gram reporter fork info every 30 seconds
dn: Mds-Software-deployment=jobmanager, Mds-Host-hn=frog.dcs.warwick.ac.uk,
    Mds-Vo-name=local, o=grid
objectclass: GlobusTop
objectclass: GlobusActiveObject
objectclass: GlobusActiveSearch
type: exec
path: /home/globus/gt2//libexec
base: globus-gram-reporter
args: -home /home/globus/gt2/ -conf /home/globus/gt2//etc/globus-job-manager.conf
      -type fork -rdn jobmanager -dmdn Mds-Host-hn=frog.dcs.warwick.ac.uk,
      Mds-Vo-name=local,o=grid
cachetime: 30
timelimit: 20
sizelimit: 20
```

grid-info-resource-register.conf

```
#
# Each LDIF record describes one registration target.  May have zero or more.
#
# Currently supported "MDSreg2" format:
#
#   dn: <LDAP add object DN>
#   regtype: mdsreg2
#   reghn: <host to send reg to>
#   regport: <port to send reg to>
#   regperiod: <how often to send reg (seconds)>
#   [service attribute/value]...
#
# where service object entries depend on the type of LDAP objct being
# published.  For MDS 2.1beta registration objects, the attributes are:
#
#   type: ldap
#   hn: <host of service being registered>
#   port: <port of service being registered>
#   rootdn: <DN suffix of service being registered>
#   ttl: <normally twice the value of regperiod>
#   timeout: <after how long should a client give up queries to service>
#   mode: cachedump
#   cachettl: <what is a reasonable time for clients to cache service>
#
#


# for default MDS 2.1 install

# register frog GRIS with frog GIIS
dn: Mds-Vo-Op-name=register, Mds-Vo-name=frog, o=grid
regtype: mdsreg2
reghn: frog.dcs.warwick.ac.uk
regport: 2135
regperiod: 600
type: ldap
```

```
hn: frog.dcs.warwick.ac.uk

port: 2135

rootdn: Mds-Vo-name=local, o=grid

ttl: 1200

#timeout: 20

timeout: 600

mode: cachedump

#cachettl: 30

cachettl: 3600
```

grid-info-site-giis.conf

The following example file is commented out and is therefore not in use in this particular instance.

```
#

# Each LDIF record describes one registration target.  May have zero or more.

#

#   Example entry:

#

#   dn: Mds-Vo-Op-name=register, Mds-Vo-name=site, o=grid

#   objectclass: Mds

#   objectclass: MdsVoOp

#   objectclass: MdsService

#   objectclass: MdsServiceLdap

#   Mds-Service-type: ldap

#   Mds-Service-hn: dc-user.isi.edu

#   Mds-Service-port: 2135

#   Mds-Service-Ldap-sizelimit: 20

#   Mds-Service-Ldap-ttl: 2000

#   Mds-Service-Ldap-cachettl: 50

#   Mds-Service-Ldap-timeout: 30

#

#

#
```

grid-info-site-policy.conf

```
#
#
#  MDS registration policy file
#
#  example:
#  objectclass: MdsRegistrationPolicy
#  policydata: (&(Mds-Service-hn=dc-*.isi.edu)(Mds-Service-port=2135))
#
#
#


# accept our own local GRIS by default
objectclass: MdsRegistrationPolicy
policydata:  (&(Mds-Service-hn=frog.dcs.warwick.ac.uk)(Mds-Service-port=2135))
```

grid-info-slapd.conf

```
schemacheck off

include /home/globus/gt2/etc/openldap/schema/core.schema

include /home/globus/gt2/etc/grid-info-resource.schema

include  /home/globus/gt2/etc/grid-info-gram-reporter.schema

pidfile /home/globus/gt2/var/resourceslapd.pid
argsfile /home/globus/gt2/var/resourceslapd.args

modulepath      /home/globus/gt2/libexec/openldap/gcc32pthr
moduleload      libback_ldif.la
moduleload      libback_giis.la

database ldif
suffix "Mds-Vo-name=local, o=grid"
conf /home/globus/gt2/etc/grid-info-resource-ldif.conf
anonymousbind   yes
access to * by * write
```

```
database        giis
suffix          "Mds-Vo-name=frog, o=grid"
conf            /home/globus/gt2/etc/grid-info-site-giis.conf
policyfile      /home/globus/gt2/etc/grid-info-site-policy.conf
anonymousbind   yes
access to * by * write
```

grid-info-deployment-comments.conf

```
# Every line of this file which does not begin with # will be
# used to generate an Mds-Service-admin-comment entry in the
# MDS software deployment object.
This is the MDS 2.4 deployment. Change this comment as you like.
```

grid-info-server-env.conf

```
#! /bin/bash

. ${GLOBUS_LOCATION}/libexec/globus-script-initializer

    if [ ! -z "${GRID_SECURITY_DIR}"      ] &&
    [ -r "${GRID_SECURITY_DIR}/ldap/ldapkey.pem" ] &&
    [ -r "${GRID_SECURITY_DIR}/ldap/ldapcert.pem" ] ; then
    X509_USER_CERT=${GRID_SECURITY_DIR}/ldap/ldapcert.pem
    X509_USER_KEY=${GRID_SECURITY_DIR}/ldap/ldapkey.pem
    elif [ -r "/etc/grid-security/ldap/ldapkey.pem" ] &&
    [ -r "/etc/grid-security/ldap/ldapcert.pem" ] ; then
    X509_USER_CERT=/etc/grid-security/ldap/ldapcert.pem
    X509_USER_KEY=/etc/grid-security/ldap/ldapkey.pem
    secconfdir="/etc/grid-security"
    elif [ -r "${GLOBUS_LOCATION}/etc/ldap/ldapkey.pem" ] &&
    [ -r "${GLOBUS_LOCATION}/etc/ldap/ldapcert.pem" ] ; then
    secconfdir="${GLOBUS_LOCATION}/etc"
    X509_USER_CERT=${GLOBUS_LOCATION}/etc/ldap/ldapcert.pem
    X509_USER_KEY=${GLOBUS_LOCATION}/etc/ldap/ldapkey.pem
    fi
```

```
# It is possible that the end is reached without any
# matching, if no certificate/key pair is found anywhere.

X509_RUN_AS_SERVER=true

GRIDMAP=${sysconfdir}/grid-mapfile

LD_LIBRARY_PATH=${GLOBUS_LOCATION}/lib:${LD_LIBRARY_PATH}
SASL_PATH=${GLOBUS_LOCATION}/lib/sasl

    export X509_USER_CERT
    export X509_USER_KEY
    export X509_RUN_AS_SERVER
    export GRIDMAP
    export LD_LIBRARY_PATH
    export SASL_PATH
```

gridftp-resource.conf

```
dn: Mds-Device-name=GridFTP,Mds-Device-Group-name=performance,
    Mds-Device-name: GridFTP
    Mds-Device-Group-name: performance
    Mds-Host-hn: frog.dcs.warwick.ac.uk
    Mds-Gridftp-gridftpurl: gsiftp://configure.me:61000
    Mds-Gridftp-loglocation: /pathto/logfile
    objectClass: MdsGridftp
```

Appendix D

MDS2 with Titan Integration

Given below are various outputs after commands are executed to search the MDS.

D.1 Output with Core Information Providers

```
[user@frog globus]$ grid-info-search -x
version: 2


#
# filter: (objectclass=*)
# requesting: ALL
#

# frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsComputer
objectClass: MdsComputerTotal
objectClass: MdsCpu
objectClass: MdsCpuCache
objectClass: MdsCpuFree
```

```
objectClass: MdsCpuSmp

objectClass: MdsCpuTotal

objectClass: MdsCpuTotalFree

objectClass: MdsFsTotal

objectClass: MdsHost

objectClass: MdsMemoryRamTotal

objectClass: MdsMemoryVmTotal

objectClass: MdsNet

objectClass: MdsNetTotal

objectClass: MdsOs

Mds-Computer-isa: i686

Mds-Computer-Total-nodeCount: 1

Mds-Computer-platform: i686

Mds-Cpu-Cache-l2kB: 512

Mds-Cpu-Free-15minX100: 073

Mds-Cpu-Free-1minX100: 073

Mds-Cpu-Free-5minX100: 061

Mds-Cpu-Smp-size: 1

Mds-Cpu-Total-Free-15minX100: 073

Mds-Cpu-Total-Free-1minX100: 073

Mds-Cpu-Total-Free-5minX100: 061

Mds-Cpu-Total-count: 1

Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm

Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2

Mds-Cpu-speedMHz: 1991

Mds-Cpu-vendor: GenuineIntel

Mds-Cpu-version: 15.2.4

Mds-Fs-Total-count: 4

Mds-Fs-Total-freeMB: 96386

Mds-Fs-Total-sizeMB: 149471

Mds-Fs-freeMB: 250

Mds-Fs-freeMB: 46379

Mds-Fs-freeMB: 49678

Mds-Fs-freeMB: 79

Mds-Fs-sizeMB: 250

Mds-Fs-sizeMB: 74006
```

```
Mds-Fs-sizeMB: 75117

Mds-Fs-sizeMB: 98

Mds-Host-hn: frog.dcs.warwick.ac.uk

Mds-Memory-Ram-Total-freeMB: 260

Mds-Memory-Ram-Total-sizeMB: 501

Mds-Memory-Ram-freeMB: 260

Mds-Memory-Ram-sizeMB: 501

Mds-Memory-Vm-Total-freeMB: 980

Mds-Memory-Vm-Total-sizeMB: 1027

Mds-Memory-Vm-freeMB: 980

Mds-Memory-Vm-sizeMB: 1027

Mds-Net-Total-count: 4

Mds-Net-addr: 127.0.0.1

Mds-Net-addr: 137.205.115.3

Mds-Net-addr: 172.16.230.1

Mds-Net-addr: 192.168.115.1

Mds-Net-name: eth0

Mds-Net-name: lo

Mds-Net-name: vmnet1

Mds-Net-name: vmnet8

Mds-Net-netaddr: 127.0.0.0/8

Mds-Net-netaddr: 137.205.115.0/25

Mds-Net-netaddr: 172.16.230.0/24

Mds-Net-netaddr: 192.168.115.0/24

Mds-Os-name: Linux

Mds-Os-release: 2.4.20-24.9

Mds-Os-version: 1 Mon Dec 1 11:35:51 EST 2003

Mds-keepto: 200312160915412

Mds-validfrom: 200312160915412

Mds-validto: 200312160915412

# processors, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-Group-name=processors, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-V
 o-name=local,o=grid

objectClass: MdsCpu

objectClass: MdsCpuSmp

objectClass: MdsCpuTotal
```

```
objectClass: MdsCpuCache

objectClass: MdsCpuFree

objectClass: MdsCpuTotalFree

objectClass: MdsDeviceGroup

Mds-Device-Group-name: processors

Mds-validfrom: 200312160943382

Mds-validto: 200312160944382

Mds-keepto: 200312190436582

Mds-Cpu-Cache-l2kB: 512

Mds-Cpu-Free-15minX100: 079

Mds-Cpu-Free-1minX100: 055

Mds-Cpu-Free-5minX100: 076

Mds-Cpu-Smp-size: 1

Mds-Cpu-Total-Free-15minX100: 079

Mds-Cpu-Total-Free-1minX100: 055

Mds-Cpu-Total-Free-5minX100: 076

Mds-Cpu-Total-count: 1

Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm

Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2

Mds-Cpu-speedMHz: 1991

Mds-Cpu-vendor: GenuineIntel

Mds-Cpu-version: 15.2.4


# cpu 0, processors, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-device-name=cpu 0, Mds-Device-Group-name=processors, Mds-Host-hn=frog.
 dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsCpu

objectClass: MdsCpuCache

Mds-Device-name: cpu 0

Mds-Cpu-vendor: GenuineIntel

Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2

Mds-Cpu-version: 15.2.4

Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm

Mds-Cpu-speedMHz: 1991
```

```
Mds-Cpu-Cache-l2kB: 512
Mds-validfrom: 200312160943382
Mds-validto: 200312160944382
Mds-keepto: 200312190436582

# memory, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-Group-name=memory, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-na
 me=local,o=grid
objectClass: MdsMemoryRamTotal
objectClass: MdsMemoryVmTotal
objectClass: MdsDeviceGroup
Mds-Device-Group-name: memory
Mds-validfrom: 200312160915412
Mds-validto: 200312161015412
Mds-keepto: 200312190409012
Mds-Memory-Ram-Total-sizeMB: 501
Mds-Memory-Ram-Total-freeMB: 259
Mds-Memory-Vm-Total-sizeMB: 1027
Mds-Memory-Vm-Total-freeMB: 980
Mds-Memory-Ram-sizeMB: 501
Mds-Memory-Ram-freeMB: 259
Mds-Memory-Vm-sizeMB: 1027
Mds-Memory-Vm-freeMB: 980

# physical memory, memory, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=physical memory, Mds-Device-Group-name=memory, Mds-Host-hn
 =frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: Mds
objectClass: MdsDevice
objectClass: MdsMemoryRam
Mds-Device-name: physical memory
Mds-Memory-Ram-sizeMB: 501
Mds-Memory-Ram-freeMB: 259
Mds-validfrom: 200312160915412
Mds-validto: 200312161015412
Mds-keepto: 200312190409012
```

```
# virtual memory, memory, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=virtual memory, Mds-Device-Group-name=memory, Mds-Host-hn=
 frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: Mds
objectClass: MdsDevice
objectClass: MdsMemoryVm
Mds-Device-name: virtual memory
Mds-Memory-Vm-sizeMB: 1027
Mds-Memory-Vm-freeMB: 980
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031219040901Z

# filesystems, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-Group-name=filesystems, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-
 Vo-name=local,o=grid
objectClass: MdsFsTotal
objectClass: MdsDeviceGroup
Mds-Device-Group-name: filesystems
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031216151541Z
Mds-Fs-Total-count: 4
Mds-Fs-Total-freeMB: 96386
Mds-Fs-Total-sizeMB: 149471
Mds-Fs-freeMB: 250
Mds-Fs-freeMB: 46379
Mds-Fs-freeMB: 49678
Mds-Fs-freeMB: 79
Mds-Fs-sizeMB: 250
Mds-Fs-sizeMB: 74006
Mds-Fs-sizeMB: 75117
Mds-Fs-sizeMB: 98

# /, filesystems, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=/, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog.dcs
 .warwick.ac.uk,Mds-Vo-name=local,o=grid
```

```
objectClass: MdsDevice
objectClass: MdsFs
Mds-Device-name: /
Mds-Fs-sizeMB: 75117
Mds-Fs-freeMB: 49678
Mds-Fs-mount: /
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031216151541Z

# /boot, filesystems, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=/boot, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog
 .dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds-Device-name: /boot
Mds-Fs-sizeMB: 98
Mds-Fs-freeMB: 79
Mds-Fs-mount: /boot
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031216151541Z

# /home, filesystems, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=/home, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog
 .dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds-Device-name: /home
Mds-Fs-sizeMB: 74006
Mds-Fs-freeMB: 46379
Mds-Fs-mount: /home
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031216151541Z

# /dev/shm, filesystems, frog.dcs.warwick.ac.uk, local, grid
```

```
dn: Mds-Device-name=/dev/shm, Mds-Device-Group-name=filesystems, Mds-Host-hn=f
 rog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds-Device-name: /dev/shm
Mds-Fs-sizeMB: 250
Mds-Fs-freeMB: 250
Mds-Fs-mount: /dev/shm
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216101541Z
Mds-keepto: 20031216151541Z

# networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-
 name=local,o=grid
objectClass: MdsNetTotal
objectClass: MdsNet
objectClass: MdsDeviceGroup
Mds-Device-Group-name: networks
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216151541Z
Mds-keepto: 20031216151541Z
Mds-Net-Total-count: 4
Mds-Net-addr: 127.0.0.1
Mds-Net-addr: 137.205.115.3
Mds-Net-addr: 172.16.230.1
Mds-Net-addr: 192.168.115.1
Mds-Net-name: eth0
Mds-Net-name: lo
Mds-Net-name: vmnet1
Mds-Net-name: vmnet8
Mds-Net-netaddr: 127.0.0.0/8
Mds-Net-netaddr: 137.205.115.0/25
Mds-Net-netaddr: 172.16.230.0/24
Mds-Net-netaddr: 192.168.115.0/24

# eth0, networks, frog.dcs.warwick.ac.uk, local, grid
```

```
dn: Mds-Device-name=eth0, Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs
 .warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: eth0
Mds-Net-name: eth0
Mds-Net-netaddr: 137.205.115.0/25
Mds-Net-addr: 137.205.115.3
Mds-validfrom: 200312160915412
Mds-validto: 200312161515412
Mds-keepto: 200312161515412

# lo, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=lo, Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs.w
 arwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: lo
Mds-Net-name: lo
Mds-Net-netaddr: 127.0.0.0/8
Mds-Net-addr: 127.0.0.1
Mds-validfrom: 200312160915412
Mds-validto: 200312161515412
Mds-keepto: 200312161515412

# vmnet1, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=vmnet1, Mds-Device-Group-name=networks, Mds-Host-hn=frog.d
 cs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: vmnet1
Mds-Net-name: vmnet1
Mds-Net-netaddr: 192.168.115.0/24
Mds-Net-addr: 192.168.115.1
Mds-validfrom: 200312160915412
Mds-validto: 200312161515412
Mds-keepto: 200312161515412
```

```
# vmnet8, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=vmnet8, Mds-Device-Group-name=networks, Mds-Host-hn=frog.d
 cs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: vmnet8
Mds-Net-name: vmnet8
Mds-Net-netaddr: 172.16.230.0/24
Mds-Net-addr: 172.16.230.1
Mds-validfrom: 20031216091541Z
Mds-validto: 20031216151541Z
Mds-keepto: 20031216151541Z

# operating system, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-deployment=operating system, Mds-Host-hn=frog.dcs.warwick.ac.
 uk,Mds-Vo-name=local,o=grid
objectClass: MdsSoftware
objectClass: MdsOs
Mds-Software-deployment: operating system
Mds-Os-name: Linux
Mds-Os-release: 2.4.20-24.9
Mds-Os-version: 1 Mon Dec 1 11:35:51 EST 2003

# MDS, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-deployment=MDS, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-nam
 e=local,o=grid
objectClass: MdsSoftware
objectClass: MdsService
objectClass: MdsServiceLdap
Mds-Software-deployment: MDS
Mds-Service-type: ldap
Mds-Service-hn: frog.dcs.warwick.ac.uk
Mds-Service-port: 2135
Mds-Service-Ldap-timeout: 30
Mds-Service-admin-contact: unspecified
Mds-Service-Executable-PID: 3392
```

```
Mds-Service-Path: /home/globus/mds2.4

Mds-Service-admin-comment: This is an MDS 2.2 deployment.

Mds-Service-Ldap-suffix: Mds-Vo-name=local,o=Grid

Mds-Service-Ldap-suffix: Mds-Vo-name=site,o=Grid

Mds-validfrom: 200312160915422

Mds-validto: 200312161515422

Mds-keepto: 200312190409022


# local, Grid

dn: Mds-Vo-name=local,o=Grid

objectClass: GlobusStub


# search result

search: 2

result: 0 Success


# numResponses: 20

# numEntries: 19
```

D.2 Custom-written Titan Information Providers

The Titan information providers which have been custom-written, are given below. `titan-get.sh`, whose script follows, is used to connect to the scheduler to retrieve the specific attribute.

titan-get.sh

```
#!/bin/sh

if [ "$#" != 3 ]

then

  echo Usage: $0 hostname port report-type >&2

  exit 1

fi
```

```
HOST=$1

PORT=$2

REPT=$3


## Run expect to contact the terminal service

expect -- <<EOF | grep "Last store:" | cut -c 13-

spawn "sh"

send "telnet $HOST $PORT\r"

expect ">"

send "connect Titan:service=reporter\r"

expect "Reporter>"

send "getlaststore $REPT\r"

expect "Reporter>"

send "logout\r"


EOF
```

access_phenotype_infoprov.pl

```
#!/usr/bin/perl


# accessing Phenotype from Titan running on frog


$a = '/home/globus/titan/titan-get.sh soda 6666 Phenotype';

chop($a);

chop($a); # because of 2 rogue characters at the end of the output

          # of the backstick command

print "dn: Mds-Software-Component=Phenotype, Mds-Software-deployment=Titan scheduler,

  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid\n";

print "objectclass: Mds\n";

print "objectclass: MdsSchedule\n";

print "Mds-Scheduler-Phenotype: ".$a;
```

access_deadline_infoprov.pl

```
#!/usr/bin/perl

# accessing Deadline from Titan on soda

$a = '/home/globus/titan/titan-get.sh soda 6666 Deadline';
chop($a);
chop($a); # because of 2 rogue characters at the end of the output
          # of the backstick command

print "dn: Mds-Software-Component=Deadline, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid\n";
print "objectclass: Mds\n";
print "objectclass: MdsSchedule\n";
print "Mds-Scheduler-Deadline: ".$a;
```

access_dominanttype_infoprov.pl

```
#!/usr/bin/perl

$a = '/home/globus/titan/titan-get.sh soda 6666 Dominant-Type';
chop($a);
chop($a); # because of 2 rogue characters at the end of the output
          # of the backstick command

print "dn: Mds-Software-Component=Dominant type, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid\n";
print "objectclass: Mds\n";
print "objectclass: MdsSchedule\n";
print "Mds-Scheduler-Dominant-Type: ".$a;
```

access_iterations_infoprov.pl

```
#!/usr/bin/perl

# Change Iterations/s below to Iterations later.

$a = '/home/globus/titan/titan-get.sh soda 6666 Iterations/s';
```

```
chop($a);
chop($a); # because of 2 rogue characters at the end of the output
          # of the backstick command

print "dn: Mds-Software-Component=Iterations, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid\n";
print "objectclass: Mds\n";
print "objectclass: MdsSchedule\n";
print "Mds-Scheduler-Iterations: ".$a;
```

access_dominance_infoprov.pl

```
#!/usr/bin/perl


$a = '/home/globus/titan/titan-get.sh soda 6666 Dominance';
chop($a);
chop($a); # because of 2 rogue characters at the end of the output
          # of the backstick command

print "dn: Mds-Software-Component=Dominance, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid\n";
print "objectclass: Mds\n";
print "objectclass: MdsSchedule\n";
print "Mds-Scheduler-Dominance: ".$a;
```

D.3 Schema for New Information Providers

```
attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.1
    NAME 'Mds-Provider-Hostname'
    DESC 'just the hostname'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.44
 )
```

```
attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.2
    NAME 'Mds-Provider-Makespan'
    DESC 'Titan makespan'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.4
    NAME 'Mds-Host-port'
    DESC 'the port number on which the slapd server is running'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.44
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.5
    NAME 'Mds-Scheduler-Type'
    DESC 'The basic mechanism for the scheduler'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.44
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.6
    NAME 'Mds-Scheduler-Nodes-Number'
    DESC 'The number of nodes under the management of the scheduler'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
)
```

```
attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.7
    NAME 'Mds-Scheduler-Version'
    DESC 'The version of the scheduler'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.44
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.8
    NAME 'Mds-Scheduler-Queue-Length'
    DESC 'The number of jobs in the scheduler queue'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.9
    NAME 'Mds-Scheduling-Algorithm'
    DESC 'Whether or not the scheduling algorithm is on'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.10
    NAME 'Mds-Scheduler-Phenotype'
    DESC 'Titan Phenotype'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
)
```

```
attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.11
    NAME 'Mds-Scheduler-Deadline'
    DESC 'Titan Deadline'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.12
    NAME 'Mds-Scheduler-Dominant-Type'
    DESC 'Titan Dominant Type'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.13
    NAME 'Mds-Scheduler-Iterations'
    DESC 'Titan number of iterations'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
 )

attributetype ( 1.3.6.1.4.1.3536.2.6.7.2.1.7.0.14
    NAME 'Mds-Scheduler-Dominance'
    DESC 'Titan dominance'
    EQUALITY integerMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
```

```
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE
 )

objectclass ( 1.3.6.1.4.1.3536.2.6.7.2.1.7
        NAME 'MdsSchedule'
        SUP 'Mds'
        AUXILIARY
        MAY ( Mds-Provider-Hostname $ Mds-Provider-Makespan  $ Mds-Host-port $
        Mds-Scheduler-Type $ Mds-Scheduler-Nodes-Number $ Mds-Scheduler-Version $
        Mds-Scheduler-Queue-Length $ Mds-Scheduling-Algorithm $ Mds-Scheduler-Phenotype $
        Mds-Scheduler-Deadline $ Mds-Scheduler-Dominant-Type $ Mds-Scheduler-Iterations $
        Mds-Scheduler-Dominance)
 )

objectclass ( 1.3.6.1.4.1.3536.2.6.7.2.1.8
        NAME 'MdsMakespan'
        SUP 'MdsSchedule'
        AUXILIARY
        MUST ( Mds-Provider-Example-Hostname $ Mds-Provider-Makespan )
 )
```

D.4 Modified grid-info-slapd.conf

```
schemacheck off

# The number of worker threads in the thread pool
threads 32

# The maximum number of entries (objects) to return
# from a search operation.
sizelimit 500

# The maximum number of seconds (in  real time) slapd will
# spend answering a search request.
timelimit 3600
```

```
include         /home/globus/mds2.4/etc/openldap/schema/core.schema
include         /home/globus/mds2.4/etc/grid-info-resource.schema
include          /home/globus/gram_reporter/warwick-scheduler.schema

pidfile         /home/globus/mds2.4/var/resourceslapd.pid
argsfile         /home/globus/mds2.4/var/resourceslapd.args

modulepath      /home/globus/mds2.4/libexec/openldap/gcc32dbgpthr
moduleload      libback_ldif.la
moduleload      libback_giis.la

database        ldif
suffix          "Mds-Vo-name=local, o=Grid"
conf            /home/globus/mds2.4/etc/grid-info-resource-ldif.conf
anonymousbind   yes
access to * by * write

database        giis
suffix          "Mds-Vo-name=site, o=Grid"
conf            /home/globus/mds2.4/etc/grid-info-site-giis.conf
policyfile      /home/globus/mds2.4/etc/grid-info-site-policy.conf
anonymousbind   yes
access to * by * write
```

D.5 Modified Output from Custom-written Information Providers

```
[user@frog etc]$ grid-info-search -x
version: 2

#
# filter: (objectclass=*)
# requesting: ALL
#

# frog.dcs.warwick.ac.uk, local, grid
```

```
dn: Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsComputer
objectClass: MdsComputerTotal
objectClass: MdsCpu
objectClass: MdsCpuCache
objectClass: MdsCpuFree
objectClass: MdsCpuSmp
objectClass: MdsCpuTotal
objectClass: MdsCpuTotalFree
objectClass: MdsFsTotal
objectClass: MdsHost
objectClass: MdsMemoryRamTotal
objectClass: MdsMemoryVmTotal
objectClass: MdsNet
objectClass: MdsNetTotal
objectClass: MdsOs
Mds-Computer-isa: i686
Mds-Computer-platform: i686
Mds-Computer-Total-nodeCount: 1
Mds-Cpu-Cache-l2kB: 512
Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
Mds-Cpu-Free-15minX100: 057
Mds-Cpu-Free-1minX100: 00
Mds-Cpu-Free-5minX100: 026
Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2
Mds-Cpu-Smp-size: 1
Mds-Cpu-speedMHz: 1991
Mds-Cpu-Total-count: 1
Mds-Cpu-Total-Free-15minX100: 057
Mds-Cpu-Total-Free-1minX100: 00
Mds-Cpu-Total-Free-5minX100: 026
Mds-Cpu-vendor: GenuineIntel
Mds-Cpu-version: 15.2.4
Mds-Fs-freeMB: 250
Mds-Fs-freeMB: 46376
Mds-Fs-freeMB: 49678
```

```
Mds-Fs-freeMB: 79

Mds-Fs-sizeMB: 250

Mds-Fs-sizeMB: 74006

Mds-Fs-sizeMB: 75117

Mds-Fs-sizeMB: 98

Mds-Fs-Total-count: 4

Mds-Fs-Total-freeMB: 96383

Mds-Fs-Total-sizeMB: 149471

Mds-Host-hn: frog.dcs.warwick.ac.uk

Mds-keepto: 20031216104741Z

Mds-Memory-Ram-freeMB: 241

Mds-Memory-Ram-sizeMB: 501

Mds-Memory-Ram-Total-freeMB: 241

Mds-Memory-Ram-Total-sizeMB: 501

Mds-Memory-Vm-freeMB: 985

Mds-Memory-Vm-sizeMB: 1027

Mds-Memory-Vm-Total-freeMB: 985

Mds-Memory-Vm-Total-sizeMB: 1027

Mds-Net-addr: 127.0.0.1

Mds-Net-addr: 137.205.115.3

Mds-Net-addr: 172.16.230.1

Mds-Net-addr: 192.168.115.1

Mds-Net-name: eth0

Mds-Net-name: lo

Mds-Net-name: vmnet1

Mds-Net-name: vmnet8

Mds-Net-netaddr: 127.0.0.0/8

Mds-Net-netaddr: 137.205.115.0/25

Mds-Net-netaddr: 172.16.230.0/24

Mds-Net-netaddr: 192.168.115.0/24

Mds-Net-Total-count: 4

Mds-Os-name: Linux

Mds-Os-release: 2.4.20-24.9

Mds-Os-version: 1 Mon Dec 1 11:35:51 EST 2003

Mds-validfrom: 20031216104741Z

Mds-validto: 20031216104741Z
```

```
# processors, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-Group-name=processors, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-V
 o-name=local,o=grid
objectClass: MdsCpu
objectClass: MdsCpuSmp
objectClass: MdsCpuTotal
objectClass: MdsCpuCache
objectClass: MdsCpuFree
objectClass: MdsCpuTotalFree
objectClass: MdsDeviceGroup
Mds-Device-Group-name: processors
Mds-validfrom: 20031216105018Z
Mds-validto: 20031216105118Z
Mds-keepto: 20031219054338Z
Mds-Cpu-Cache-l2kB: 512
Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
Mds-Cpu-Free-15minX100: 046
Mds-Cpu-Free-1minX100: 00
Mds-Cpu-Free-5minX100: 009
Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2
Mds-Cpu-Smp-size: 1
Mds-Cpu-speedMHz: 1991
Mds-Cpu-Total-count: 1
Mds-Cpu-Total-Free-15minX100: 046
Mds-Cpu-Total-Free-1minX100: 00
Mds-Cpu-Total-Free-5minX100: 009
Mds-Cpu-vendor: GenuineIntel
Mds-Cpu-version: 15.2.4

# cpu 0, processors, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-device-name=cpu 0, Mds-Device-Group-name=processors, Mds-Host-hn=frog.
 dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsCpu
objectClass: MdsCpuCache
Mds-Device-name: cpu 0
```

```
Mds-Cpu-vendor: GenuineIntel

Mds-Cpu-model: Intel(R) Pentium(R) 4 CPU 2

Mds-Cpu-version: 15.2.4

Mds-Cpu-features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo

 v pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm

Mds-Cpu-speedMHz: 1991

Mds-Cpu-Cache-l2kB: 512

Mds-validfrom: 20031216105018Z

Mds-validto: 20031216105118Z

Mds-keepto: 20031219054338Z


# memory, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-Group-name=memory, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-na

 me=local,o=grid

objectClass: MdsMemoryRamTotal

objectClass: MdsMemoryVmTotal

objectClass: MdsDeviceGroup

Mds-Device-Group-name: memory

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031219054104Z

Mds-Memory-Ram-Total-sizeMB: 501

Mds-Memory-Ram-Total-freeMB: 241

Mds-Memory-Vm-Total-sizeMB: 1027

Mds-Memory-Vm-Total-freeMB: 985

Mds-Memory-Ram-sizeMB: 501

Mds-Memory-Ram-freeMB: 241

Mds-Memory-Vm-sizeMB: 1027

Mds-Memory-Vm-freeMB: 985


# physical memory, memory, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=physical memory, Mds-Device-Group-name=memory, Mds-Host-hn

 =frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: Mds

objectClass: MdsDevice

objectClass: MdsMemoryRam

Mds-Device-name: physical memory
```

```
Mds-Memory-Ram-sizeMB: 501

Mds-Memory-Ram-freeMB: 241

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031219054104Z


# virtual memory, memory, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=virtual memory, Mds-Device-Group-name=memory, Mds-Host-hn=
 frog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: Mds

objectClass: MdsDevice

objectClass: MdsMemoryVm

Mds-Device-name: virtual memory

Mds-Memory-Vm-sizeMB: 1027

Mds-Memory-Vm-freeMB: 985

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031219054104Z


# filesystems, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-Group-name=filesystems, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-
 Vo-name=local,o=grid

objectClass: MdsFsTotal

objectClass: MdsDeviceGroup

Mds-Device-Group-name: filesystems

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031216164744Z

Mds-Fs-freeMB: 250

Mds-Fs-freeMB: 46376

Mds-Fs-freeMB: 49678

Mds-Fs-freeMB: 79

Mds-Fs-sizeMB: 250

Mds-Fs-sizeMB: 74006

Mds-Fs-sizeMB: 75117

Mds-Fs-sizeMB: 98

Mds-Fs-Total-count: 4
```

```
Mds-Fs-Total-freeMB: 96383

Mds-Fs-Total-sizeMB: 149471


# /, filesystems, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=/, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog.dcs
 .warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsFs

Mds-Device-name: /

Mds-Fs-sizeMB: 75117

Mds-Fs-freeMB: 49678

Mds-Fs-mount: /

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031216164744Z


# /boot, filesystems, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=/boot, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog
 .dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsFs

Mds-Device-name: /boot

Mds-Fs-sizeMB: 98

Mds-Fs-freeMB: 79

Mds-Fs-mount: /boot

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031216164744Z


# /home, filesystems, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=/home, Mds-Device-Group-name=filesystems, Mds-Host-hn=frog
 .dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsFs

Mds-Device-name: /home

Mds-Fs-sizeMB: 74006

Mds-Fs-freeMB: 46376
```

```
Mds-Fs-mount: /home

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031216164744Z


# /dev/shm, filesystems, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=/dev/shm, Mds-Device-Group-name=filesystems, Mds-Host-hn=f
 rog.dcs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsFs

Mds-Device-name: /dev/shm

Mds-Fs-sizeMB: 250

Mds-Fs-freeMB: 250

Mds-Fs-mount: /dev/shm

Mds-validfrom: 20031216104744Z

Mds-validto: 20031216114744Z

Mds-keepto: 20031216164744Z


# networks, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-
 name=local,o=grid

objectClass: MdsNetTotal

objectClass: MdsNet

objectClass: MdsDeviceGroup

Mds-Device-Group-name: networks

Mds-validfrom: 20031216104745Z

Mds-validto: 20031216164745Z

Mds-keepto: 20031216164745Z

Mds-Net-addr: 127.0.0.1

Mds-Net-addr: 137.205.115.3

Mds-Net-addr: 172.16.230.1

Mds-Net-addr: 192.168.115.1

Mds-Net-name: eth0

Mds-Net-name: lo

Mds-Net-name: vmnet1

Mds-Net-name: vmnet8

Mds-Net-netaddr: 127.0.0.0/8
```

```
Mds-Net-netaddr: 137.205.115.0/25
Mds-Net-netaddr: 172.16.230.0/24
Mds-Net-netaddr: 192.168.115.0/24
Mds-Net-Total-count: 4

# eth0, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=eth0, Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs
 .warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: eth0
Mds-Net-name: eth0
Mds-Net-netaddr: 137.205.115.0/25
Mds-Net-addr: 137.205.115.3
Mds-validfrom: 200312161047452
Mds-validto: 200312161647452
Mds-keepto: 200312161647452

# lo, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=lo, Mds-Device-Group-name=networks, Mds-Host-hn=frog.dcs.w
 arwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: lo
Mds-Net-name: lo
Mds-Net-netaddr: 127.0.0.0/8
Mds-Net-addr: 127.0.0.1
Mds-validfrom: 200312161047452
Mds-validto: 200312161647452
Mds-keepto: 200312161647452

# vmnet1, networks, frog.dcs.warwick.ac.uk, local, grid
dn: Mds-Device-name=vmnet1, Mds-Device-Group-name=networks, Mds-Host-hn=frog.d
 cs.warwick.ac.uk,Mds-Vo-name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds-Device-name: vmnet1
```

```
Mds-Net-name: vmnet1

Mds-Net-netaddr: 192.168.115.0/24

Mds-Net-addr: 192.168.115.1

Mds-validfrom: 20031216104745Z

Mds-validto: 200312161647 45Z

Mds-keepto: 200312161647 45Z


# vmnet8, networks, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Device-name=vmnet8, Mds-Device-Group-name=networks, Mds-Host-hn=frog.d
 cs.warwick.ac.uk,Mds-Vo-name=local,o=grid

objectClass: MdsDevice

objectClass: MdsNet

Mds-Device-name: vmnet8

Mds-Net-name: vmnet8

Mds-Net-netaddr: 172.16.230.0/24

Mds-Net-addr: 172.16.230.1

Mds-validfrom: 200312161047 45Z

Mds-validto: 200312161647 45Z

Mds-keepto: 200312161647 45Z


# operating system, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Software-deployment=operating system, Mds-Host-hn=frog.dcs.warwick.ac.
 uk,Mds-Vo-name=local,o=grid

objectClass: MdsSoftware

objectClass: MdsOs

Mds-Software-deployment: operating system

Mds-Os-name: Linux

Mds-Os-release: 2.4.20-24.9

Mds-Os-version: 1 Mon Dec 1 11:35:51 EST 2003


# MDS, frog.dcs.warwick.ac.uk, local, grid

dn: Mds-Software-deployment=MDS, Mds-Host-hn=frog.dcs.warwick.ac.uk,Mds-Vo-nam
 e=local,o=grid

objectClass: MdsSoftware

objectClass: MdsService

objectClass: MdsServiceLdap

Mds-Software-deployment: MDS
```

```
Mds-Service-type: ldap
Mds-Service-hn: frog.dcs.warwick.ac.uk
Mds-Service-port: 2135
Mds-Service-Ldap-timeout: 30
Mds-Service-admin-contact: unspecified
Mds-Service-Executable-PID: 6251
Mds-Service-Path: /home/globus/mds2.4
Mds-Service-admin-comment: This is an MDS2.4 deployment, with additional GRAM
 information providers. (c)
Mds-Service-admin-comment: Helene Lim, Dec 2003.
Mds-Service-Ldap-suffix: Mds-Vo-name=local,o=Grid
Mds-Service-Ldap-suffix: Mds-Vo-name=site,o=Grid
Mds-validfrom: 20031216104746Z
Mds-validto: 20031216164746Z
Mds-keepto: 20031219054106Z

# Phenotype, Titan scheduler, soda.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-Component=Phenotype, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSchedule
Mds-Scheduler-Phenotype: 335499.75

# Deadline, Titan scheduler, soda.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-Component=Deadline, Mds-Software-deployment=Titan scheduler,
 Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSchedule
Mds-Scheduler-Deadline: 31844.0

# Dominant type, Titan scheduler, soda.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-Component=Dominant type, Mds-Software-deployment=Titan schedu
 ler, Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSchedule
Mds-Scheduler-Dominant-Type: 0.0
```

```
# Iterations, Titan scheduler, soda.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-Component=Iterations, Mds-Software-deployment=Titan scheduler
 , Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSchedule
Mds-Scheduler-Iterations: 55.370983

# Dominance, Titan scheduler, soda.dcs.warwick.ac.uk, local, grid
dn: Mds-Software-Component=Dominance, Mds-Software-deployment=Titan scheduler,
  Mds-Host-hn=soda.dcs.warwick.ac.uk, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSchedule
Mds-Scheduler-Dominance: 22937.32

# local, Grid
dn: Mds-Vo-name=local,o=Grid
objectClass: GlobusStub

# search result
search: 2
result: 0 Success

# numResponses: 25
# numEntries: 24
```

Appendix E

Graphs of Percentage of CPU Idleness

Below are graphs showing the distributions of the CPU idleness on the Index Service host for each of the different experiments carried out in Section 6.3.

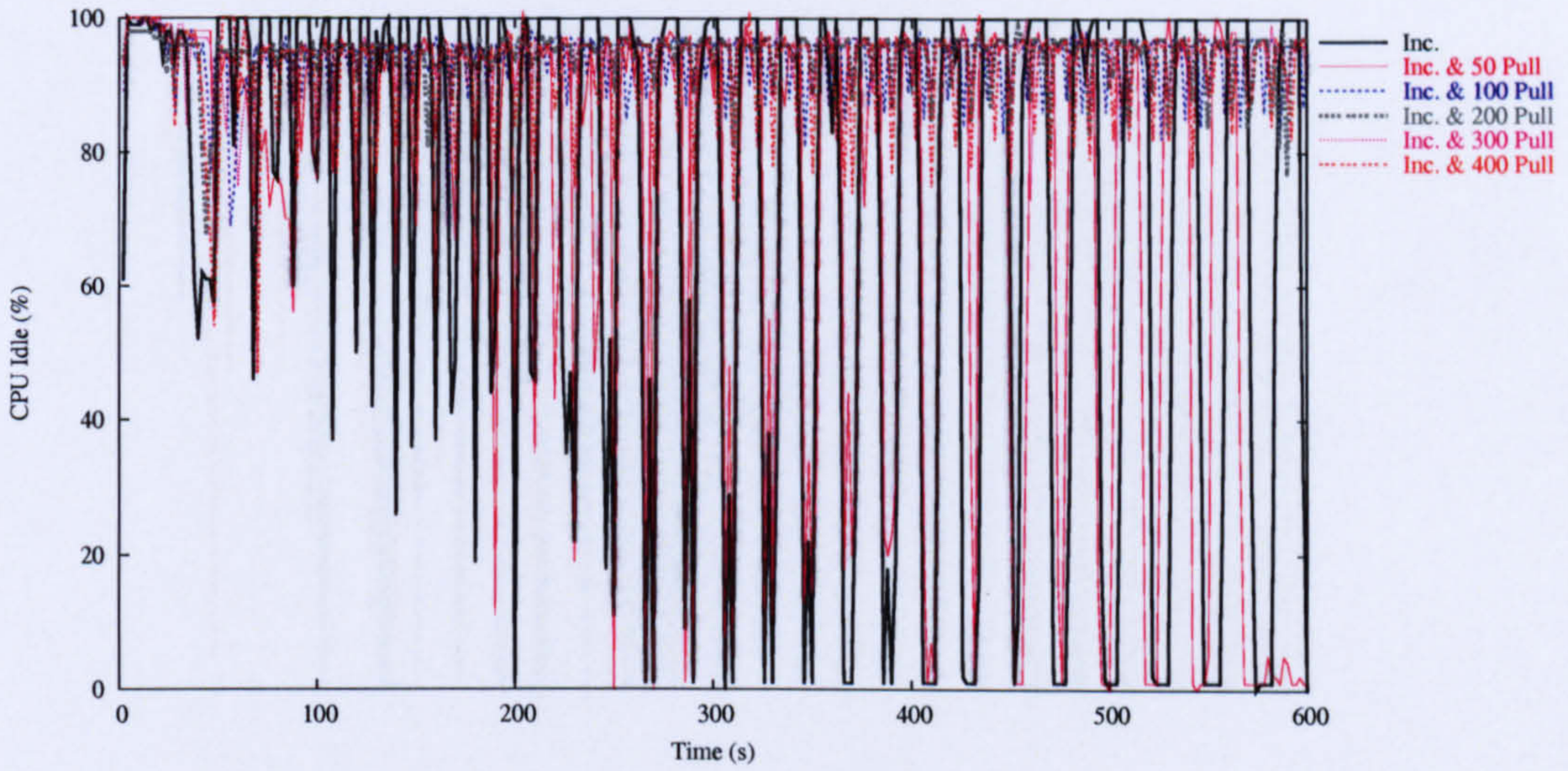

Figure E.1: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents with a 5 s wait time.

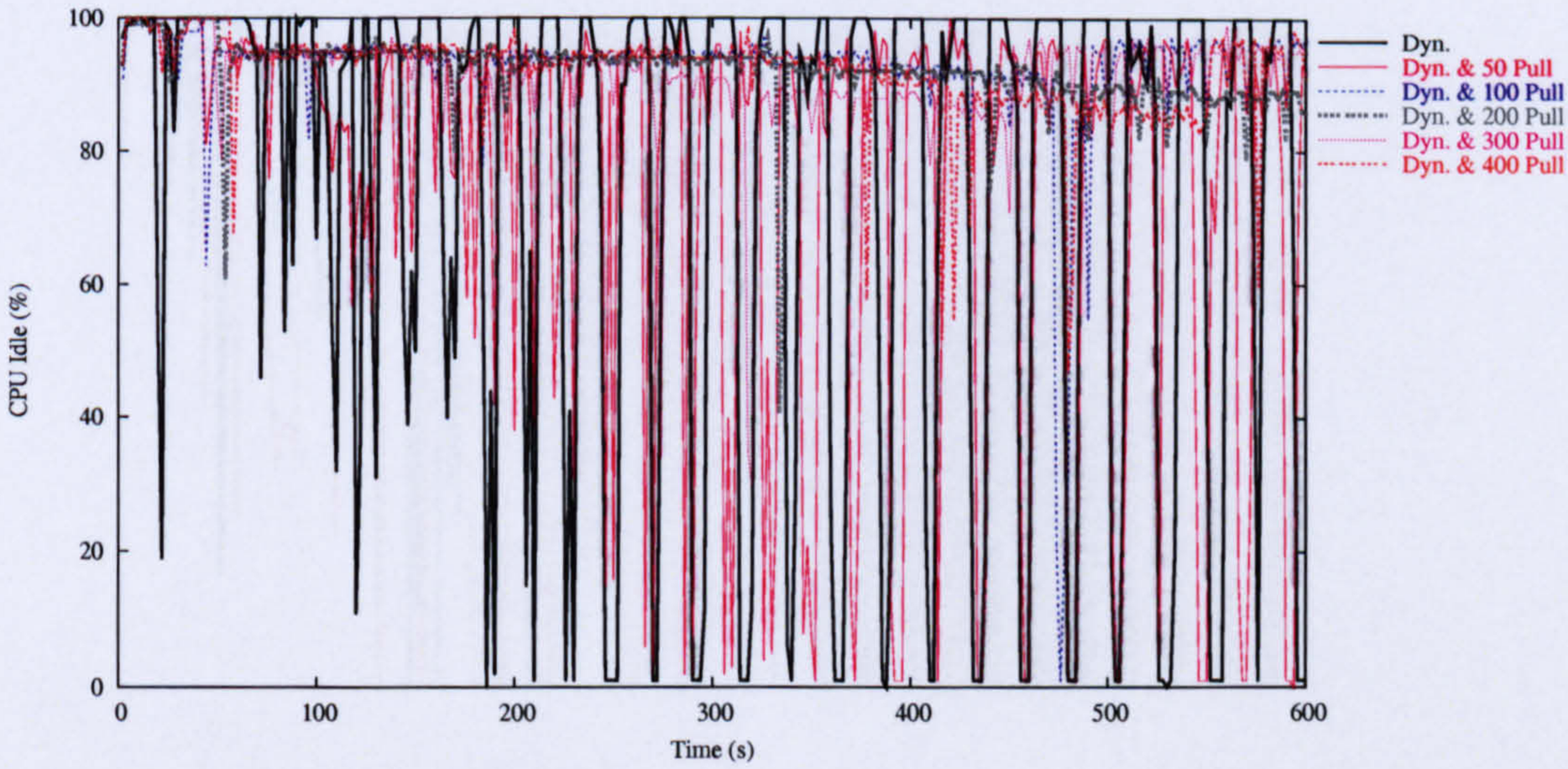

Figure E.2: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents with a 5 s wait time.

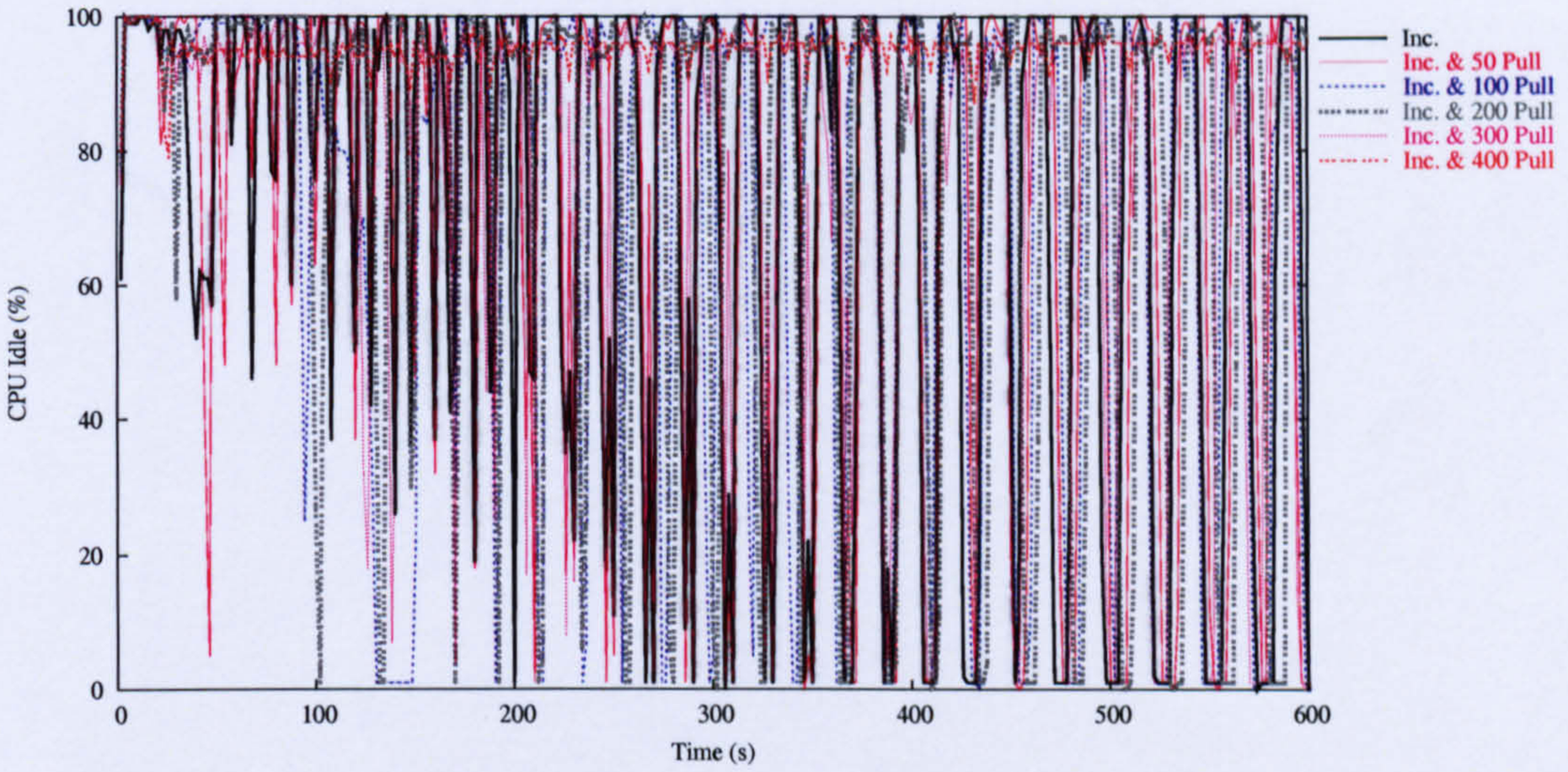

Figure E.3: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents with a 30 s wait time.

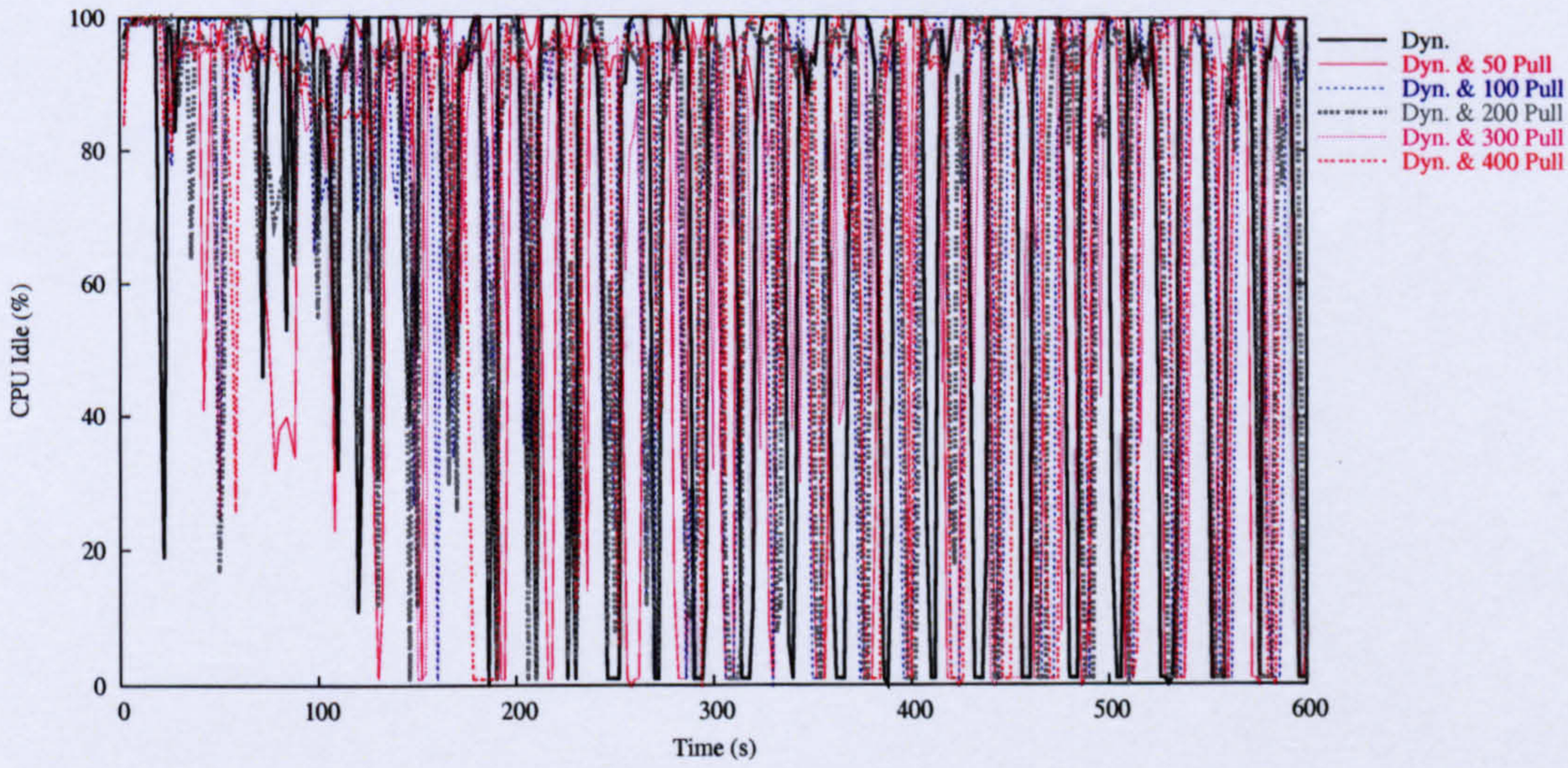

Figure E.4: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents with a 30 s wait time.

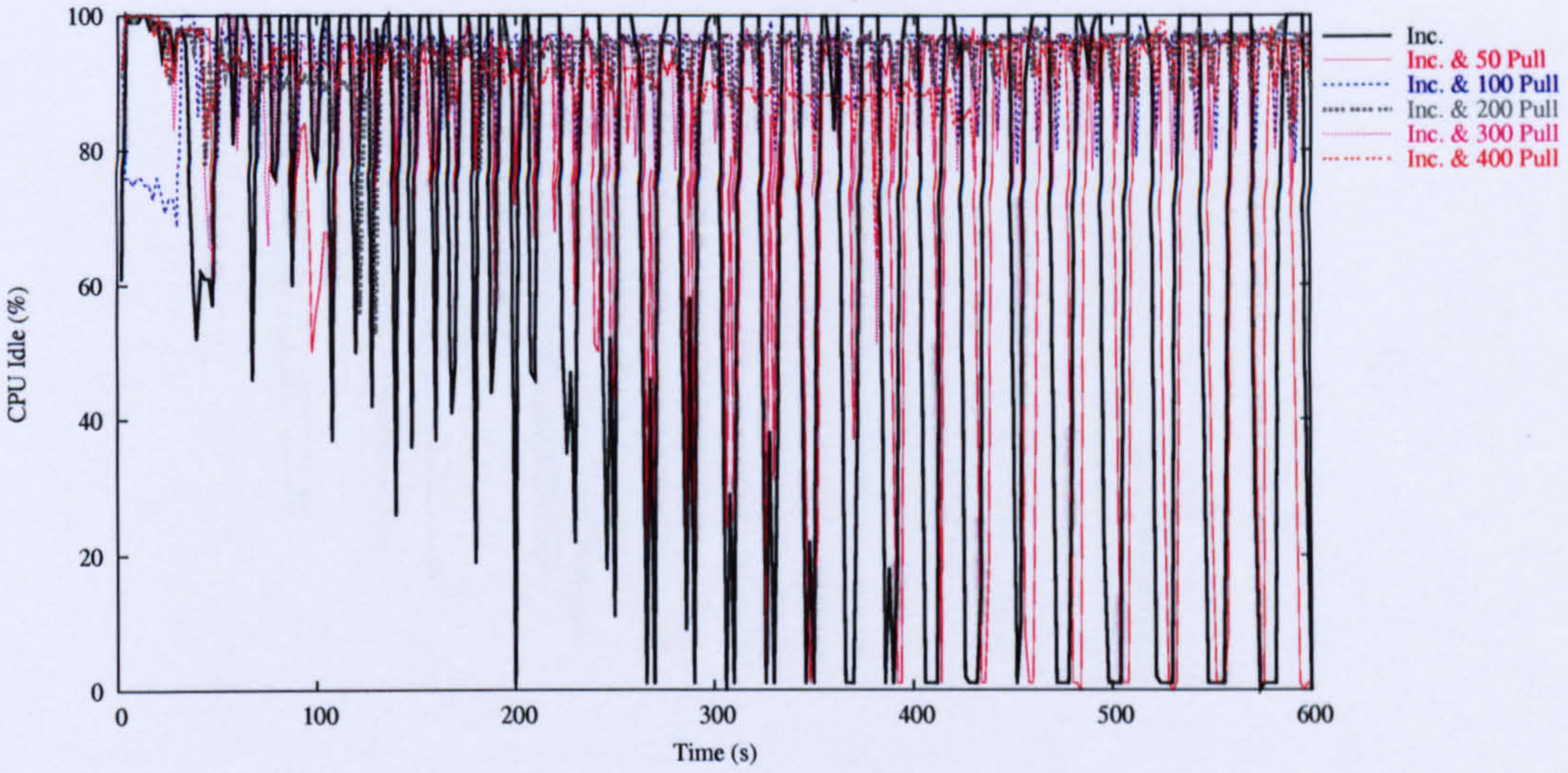

Figure E.5: Percentage of CPU idleness for self-adaptive push agents (increasing workload) and pull agents.

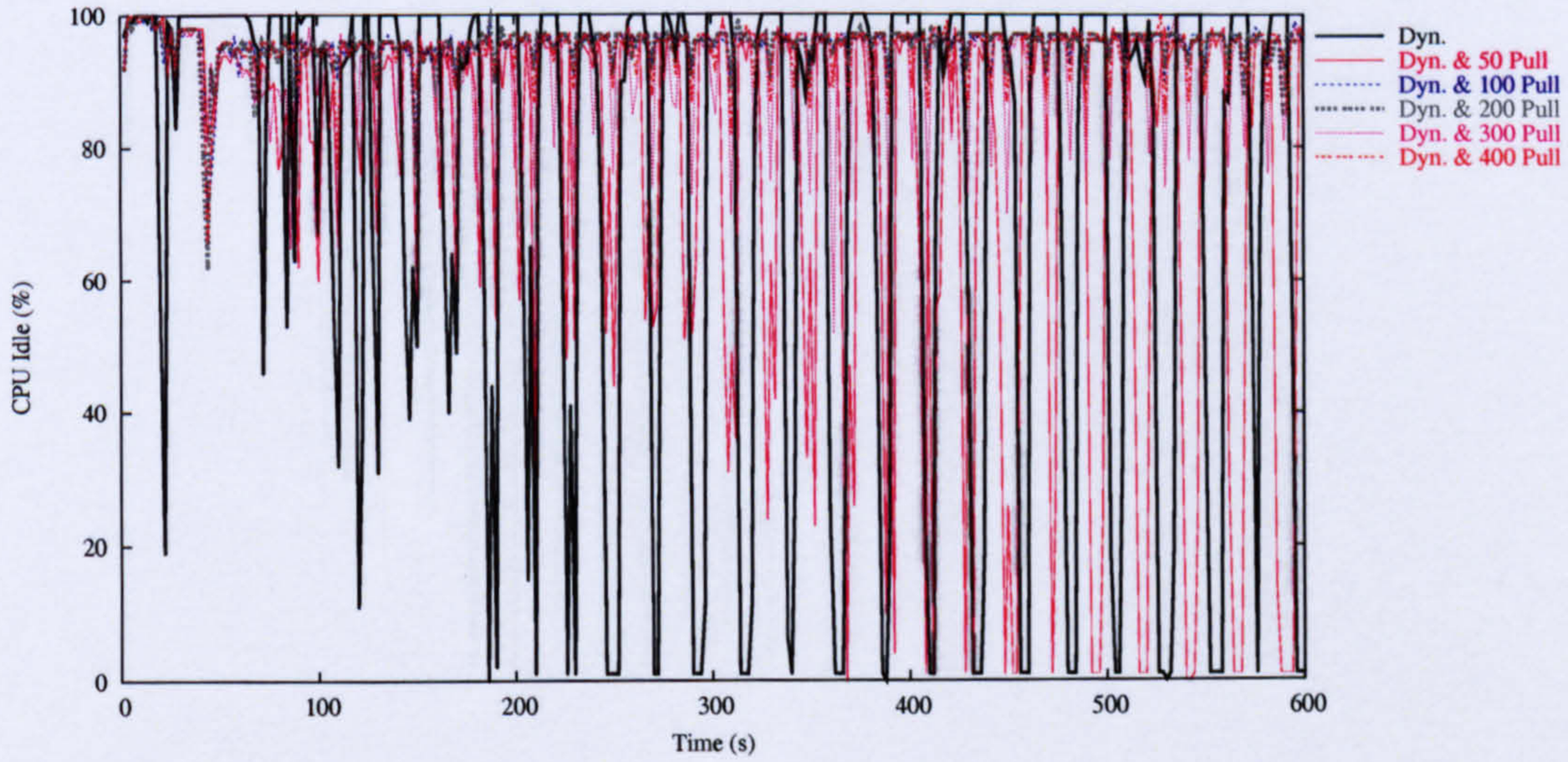

Figure E.6: Percentage of CPU idleness for self-adaptive push agents (dynamic workload) and pull agents.

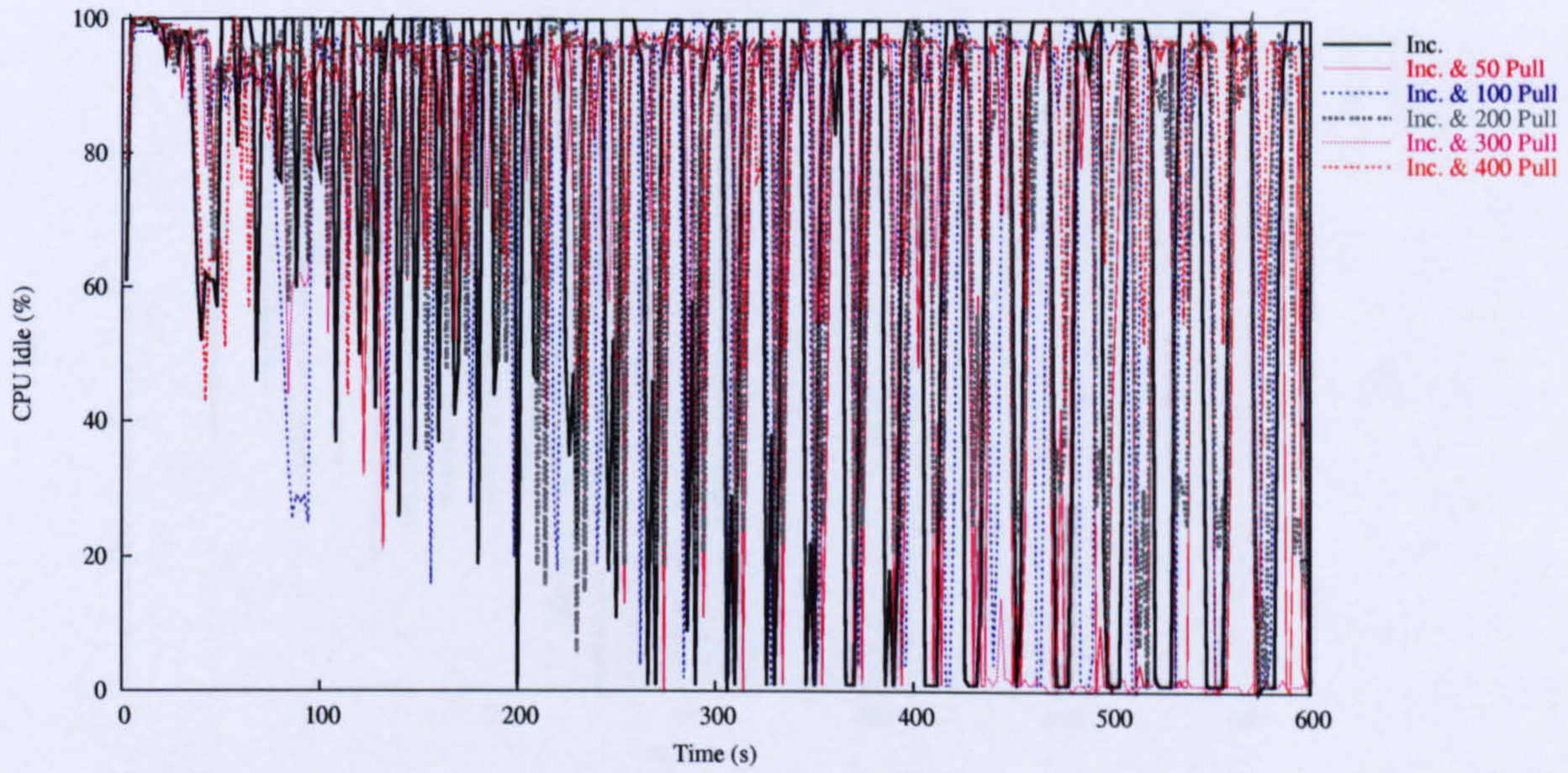

Figure E.7: Percentage of CPU idleness for the admission control of pull-based queries with push-based (increasing workload) agents.

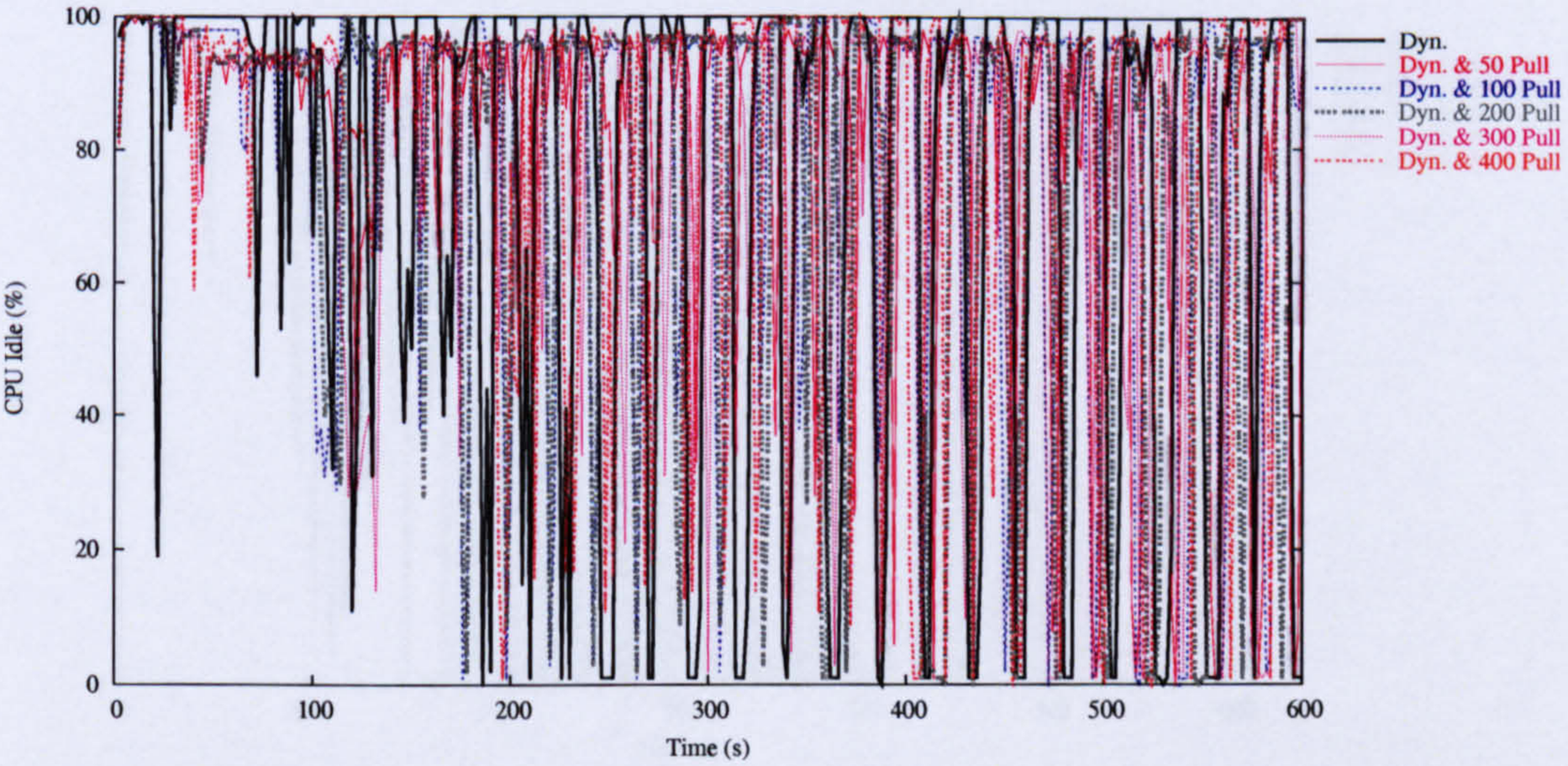

Figure E.8: Percentage of CPU idleness for the admission control of pull-based queries with push-based (dynamic workload) agents.

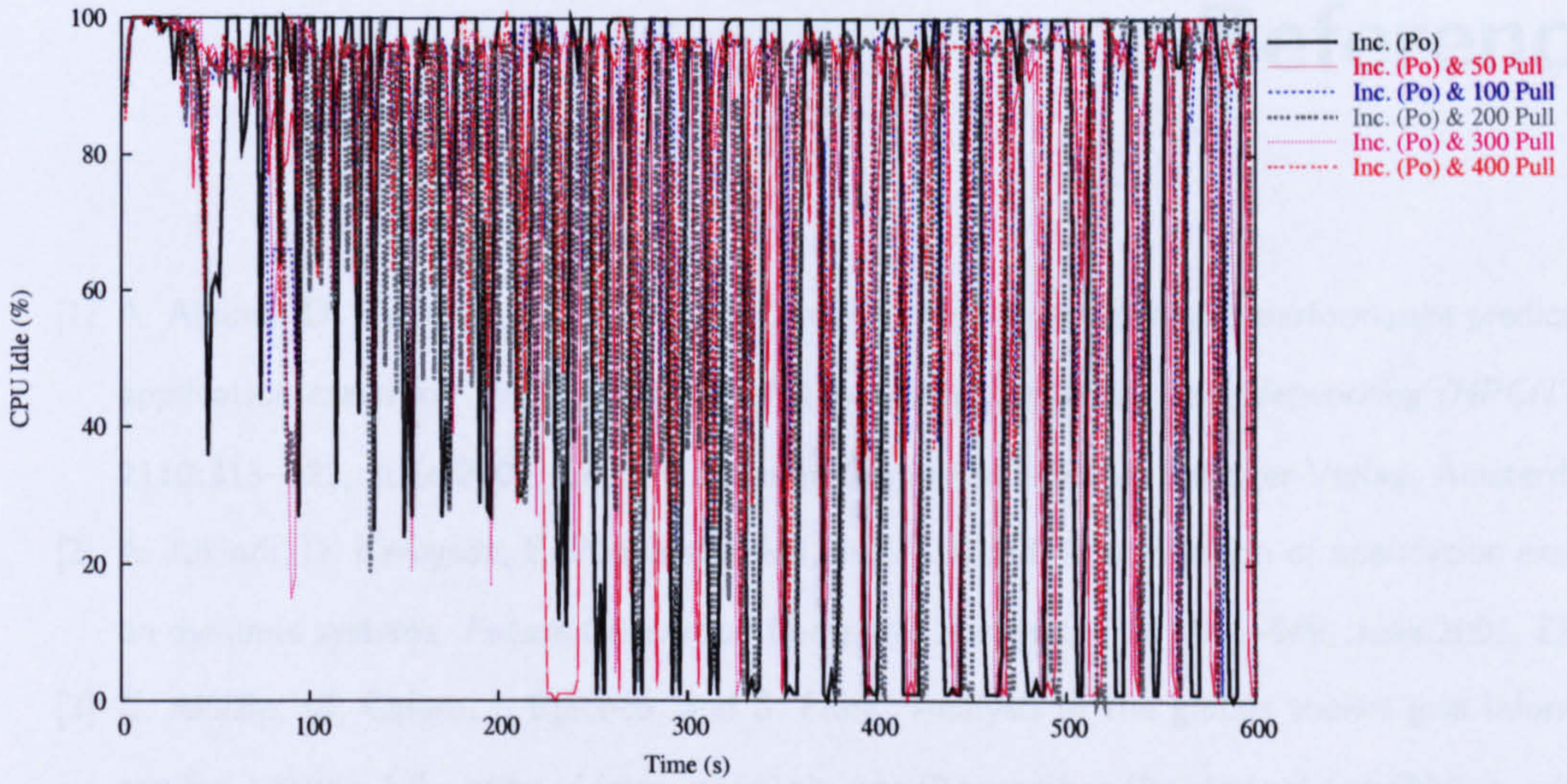

Figure E.9: Percentage of CPU idleness for the admission control of pull-based queries with push-based (increasing Poisson workload) agents.

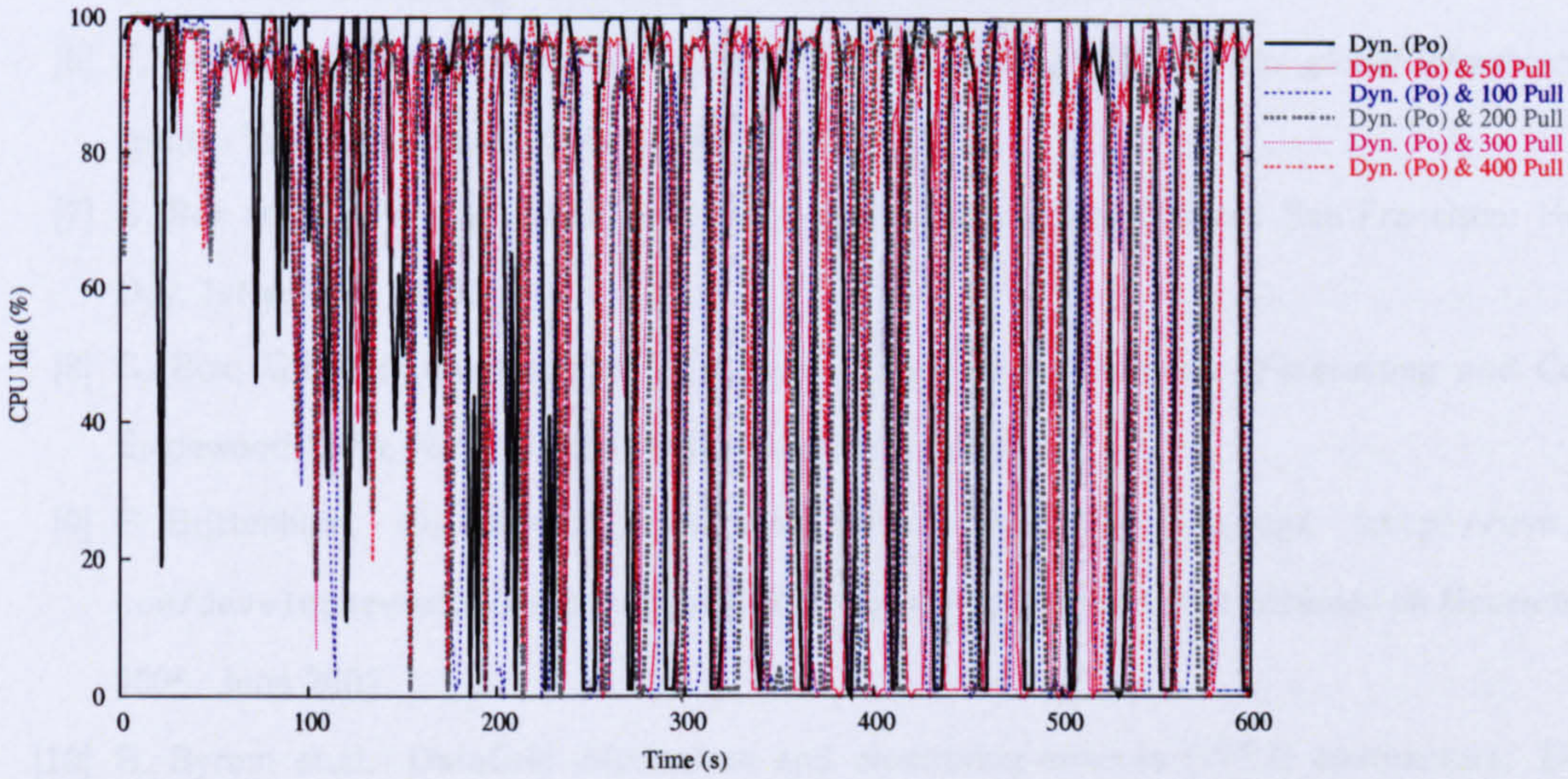

Figure E.10: Percentage of CPU idleness for the admission control of pull-based queries with push-based (dynamic Poisson workload) agents.

References

[1] A. Alkindi, D. Kerbyson, and G. Nudd. Dynamic instrumentation and performance prediction of application execution. *In Proc. of High-Performance Computing and Networking (HPCN'2001)*, 2110:313–323, June 2001. Lecture Notes in Computer Science, Springer-Verlag, Amsterdam.

[2] A. Alkindi, D. Kerbyson, E. Papaefstathiou, and G. Nudd. Optimisation of application execution on dynamic systems. *Future Generation Computer Systems*, 17(8):941–949, June 2001. Elsevier.

[3] G. Aloisio, M. Cafaro, I. Epicoco, and S. Fiore. Analysis of the globus toolkit grid information service, version 1.0. `http://www.gridlab.org/Resources/Deliverables/D10.1.pdf`, last accessed on September 5, 2005.

[4] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. The Jini specification. *Addison-Wesley, Reading, MA, USA*, 1999.

[5] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level Grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, Winter 2001.

[6] F. Berman, G. Fox, and A. Hey, editors. *Grid Computing: Making the global infrastructure a reality*. Wiley, Chichester, 2003. page 557.

[7] G. Box and G. Jenkins. *Time Series Analysis Forecasting and Control*. San Francisco: Holden-Day, 1st edition, 1970.

[8] G. Box, G.E.Pand Jenkins and G. Reinsell. *Time Series Analysis: Forecasting and Control*. Englewood Cliffs, N.J.: Prentice-Hall, 3rd edition, 1994.

[9] P. Brittenham. An overview of the Web Services Inspection Language. `http://www.ibm.com/developerworks/webservices/library/ws-wsilover/`, last accessed on November 10, 2005, June 2002.

[10] B. Byrom et.al. DataGrid information and monitoring services (WP3) architecture: Design, requirements and evaluation criteria. *WP3 Technical Report, DataGrid*, 2002.

[11] J. Cao, S. Jarvis, S. Saini, D. Kerbyson, and G. Nudd. ARMS: An Agent-based resource management system for Grid computing. *Scientific Programming Special Issue on Grid Computing*, 2002.

[12] J. Cao, D. Kerbyson, and G. Nudd. Use of agent-based service discovery for resource management in metacomputing environment. *In Proc. 7th International Euro-Par Conference, Manchester, UK*, pages 882–886, August 2001. Lecture Notes in Computer Science, 2150, Springer-Verlag.

[13] J. Cao, D. Spooner, J. Turner, S. Jarvis, D. Kerbyson, S. Saini, and G. Nudd. Agent-based resource management for Grid computing. *In Proc. 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), Berlin, Germany*, pages 350–351, May 2002.

[14] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[15] J. Clark. XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`, W3C Recommendation, last accessed on January 20, 2006, November 16 1999.

[16] Condor classified advertisements. `http://www.cs.wisc.edu/condor/classad/`, last accessed on November 7, 2005.

[17] D. Connolly, F. van Harmelen, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. DAML+OIL Reference Description. March 2001. W3C.

[18] A. Cooke, A.Gray, L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, and et al. R-GMA: An Information Integration System for Grid Monitoring. *Proc, 11th International Conference on Cooperative Information Systems*, 2003. Lecture Notes in Computer Science, Springer-Verlag Heidelberg.

[19] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution, Version 1.1, 03/05/2004. `http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf`, last accessed on September 5, 2005.

[20] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework: Version 1.0, 03/05/2004. `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`, last accessed on September 23, 2005.

[21] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. *In Proc. 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), San Francisco, CA*, pages 181–194, August 7-9, 2001. ISBN 0-7695-1296-8, IEEE Computer Society.

[22] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. Web Ontology Language (OWL) Reference version 1.0. 2002. W3C.

[23] Distributed Management Task Force. `http://www.dmtf.org/`, last accessed on November 12, 2005.

[24] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. DiPerF: An Automated DIstributed PERformance Testing Framework. *In Proc. 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), Pittsburgh, PA*, pages 289–296, November 8, 2004.

[25] N. Dushay, J. C. French, and C. Lagoze. Predicting indexer performance in a distributed digital library. *In Proc. 3rd European Conference on Research and Advanced Technology for Digital Libraries (ECDL'99), Paris, France*, September 1999.

[26] J. Dyson, N. Griffiths, H. Lim Choi Keung, S. Jarvis, and G. Nudd. Trusting agents for Grid Computing. *Special Track, held as part of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2004), The Hague, Netherlands*, October 10-13, 2004.

[27] G. Eisenhauer, F. Bustamante, and K. Schwan. Event services in high performance systems. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 4(3):243–252, July 2001.

[28] Enabling Grids for e-Science in Europe. `http://public.eu-egee.org/`, last accessed on November 9, 2005.

[29] Event service specification, Version 1.1, March 2001. `http://www.omg.org/docs/formal/01-03-01.pdf`, last accessed on November 10, 2005.

[30] S. Fitzgerald, I. Foster, C. Kesselman, G. Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. *In Proc. 6th IEEE International Symposium on High-Performance Distributed Computing (HPDC-6), Portland, Oregon*, pages 365–375, August 5-8, 1997. IEEE Press.

[31] I. Foster. Globus Toolkit version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing (NPC'2005), Beijing, China*, pages 2–13, November 30 - December 3 2005. Lecture Notes in Computer Science 3779, ISBN 3-540-29810, Springer-Verlag.

[32] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1998. ISBN 1-5586-0475-8.

[33] I. Foster and C. Kesselman. A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–129, 1998.

[34] I. Foster and C. Kesselman, editors. *The Grid2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 2004. ISBN 1-5586-0933-4.

[35] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An Open Grid Services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22, 2002.

[36] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed systems integration. *IEEE Computer*, 35(6):37–46, 2002.

[37] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl.*, 15(3):200–222, 2001.

[38] B. Gaidioz, R. Wolski, and B. Tourancheau. Synchronizing network probes to avoid measurement intrusiveness with the Network Weather Service. *In Proc. 9th IEEE International Symposium on High-Performance Distributed Computing (HPDC-9), Pittsburgh, Pennsylvania*, pages 147–154, August 1-4, 2000. IEEE Computer Society.

[39] Ganglia monitoring system. `http://ganglia.sourceforge.net/`. last accessed on November 9, 2005.

[40] Global Grid Forum. `http://www.ggf.org`, last accessed on November 8, 2005.

[41] Globus: Extending GRIS Functionality. `http://www-fp.globus.org/mds/extending-gris.html`, last accessed on September 5, 2005.

[42] Globus Toolkit 2.2, MDS Technology Brief, Draft 4, January 30, 2003. `http://www.globus.org/toolkit/docs/2.4/mds/mdstechnologybrief_draft4.pdf`, last accessed on July 19, 2005.

[43] Globus Toolkit Information Services: Monitoring and Discovery System (MDS). `http://www.globus.org/toolkit/mds/`, last accessed on November 3, 2005.

[44] GLUE Information Model. `http://infnforge.cnaf.infn.it/glueinfomodel`, last accessed on September 5, 2005.

[45] GLUE Schema. `http://www.globus.org/toolkit/mds/glueschemalink.html`, last accessed on September 5, 2005.

[46] GLUE Schema Specification, Draft 8, July 22, 2005. `http://infnforge.cnaf.infn.it/glueinfomodel/uploads/Spec/GLUEInfoModel%_1_2_draft_8.pdf`, last accessed on September 2, 2005.

[47] GT3 OGSA 3.0.2 APIs. `http://www-unix.globus.org/toolkit/3.0/ogsa/impl/java/build/javadocs/`, last accessed on September 5, 2005.

[48] Hawkeye: A monitoring and management tool for distributed systems. `http://www.cs.wisc.edu/condor/hawkeye/`, last accessed on November 7, 2005.

[49] P. Horn. Autonomic Computing: IBM's perspective on the state of information technology. *IBM Corporation*, 15 October 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, last accessed on September 6, 2005.

[50] T. Howes and M. Smith. *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol.* Macmillan Technical Publishing, 1997.

[51] Information Services in the Globus Toolkit 3.0 Release. `http://www.globus.org/toolkit/docs/3.0/mds`, last accessed on September 5, 2005.

[52] International Virtual Data Grid Laboratory (iVDGL). `http://www.ivdgl.org/`, last accessed on September 1, 2005.

[53] S. Jarvis, D. Spooner, H. Lim Choi Keung, J. Cao, S. Saini, and G. Nudd. Performance prediction and its use in parallel and distributed computing systems computing systems. *In International*

Workshop on Performance Modelling, Evaluation and Optimisation of Parallel and Distributed Systems, held as part of the 17th International Parallel and Distributed Processing Symposium (IPDPS'2003), Nice, France, page 276, April 22-26 2003. ISBN 0-7695-1926-1, ACM/IEEE Computer Society.

[54] S. Jarvis, D. Spooner, H. Lim Choi Keung, J. Cao, S. Saini, and G. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems, Special Issue on System Performance Analysis and Evaluation*, 2006.

[55] S. Jarvis, D. Spooner, H. Lim Choi Keung, J. Dyson, L. Zhao, and G. Nudd. Performance-based middleware services for Grid Computing computing systems. *In 5th International Workshop on Active Middleware Services (AMS'2003), held as part of the 12th IEEE International Symposium on High-Performance Distributed Computing (HPDC-12), Seattle, USA*, pages 151–159, June 25 2003. ISBN 0-7695-1983-0, IEEE Computer Society.

[56] Java class API OutputStream. `http://java.sun.com/j2se/1.4.2/docs/api/java/io/OutputStream.html`, last accessed on September 5, 2005.

[57] Java class API TimerTask. `http://java.sun.com/j2se/1.4.2/docs/api/java/util/TimerTask.html`, last accessed on September 5, 2005.

[58] Jini Network Technology. `http://www.sun.com/jini`, last accessed on November 10, 2005.

[59] Joint research activity 1: Information and monitoring, Enabling Grids for e-Science in Europe. `http://hepunx.rl.ac.uk/egee/jra1-uk/`, last accessed on November 9, 2005.

[60] W. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. *In Proc. 8th IEEE International Symposium on High-Performance Distributed Computing (HPDC-8), Redondo Beach, California*, pages 197–204, August 3-6, 1999. ISBN 0-7803-5681-0, IEEE Computer Society Press.

[61] D. Kerbyson, J. Harper, E. Papaefstathiou, D. Wilcox, and G. Nudd. Use of performance technology for the management of distributed systems. *In Proc. European Conference on Parallel Computing (Euro-Par 2000), Munich, Germany*, August 29 - 1 September, 2000. Lecture Notes of Computer Science, Springer-Verlag.

[62] D. Kerbyson, E. Papaefstathiou, and G. Nudd. Application execution steering using on-the-fly performance prediction. *In Proc. International Conference and Exhibition on High-Performance Computing and Networking (HPCN'1998)*, 1401:718–727, April 1998. Lecture Notes in Computer Science, Springer-Verlag.

[63] M. Koivunen and E. Miller. W3C Semantic Web activity. *Semantic Web Kick-off Seminar in Finland, part of Semantic Web in Finland effort, Helsinki, Finland*, November 2001.

[64] Lightweight Directory Access Protocol (v3). `http://www.ietf.org/rfc/rfc2251.txt`, last accessed on September 3, 2005.

[65] Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions. `ftp://ftp.isi.edu/in-notes/rfc2252.txt`, last accessed on July 21, 2005.

[66] H. Lim Choi Keung, J. Cao, D. Spooner, S. Jarvis, and G. Nudd. Grid Information Services using software agents. *In Proc. 18th Annual UK Performance Engineering Workshop (UKPEW'2002), University of Glasgow, UK*, pages 187–198, July 10-11, 2002.

[67] H. Lim Choi Keung, J. Dyson, S. Jarvis, and G. Nudd. Performance evaluation of a Grid resource monitoring and discovery service. *IEE Proc.-Software*, 150(4):243–251, August 2003.

[68] H. Lim Choi Keung, J. Dyson, S. Jarvis, and G. Nudd. Predicting the performance of Globus monitoring and discovery service (MDS-2) queries. *In Proc. 4th ACM/IEEE International Workshop on Grid Computing (GRID'2003), held as part of SuperComputing 2003, Phoenix, Arizona*, pages 176–183, November 17, 2003.

[69] H. Lim Choi Keung, J. Dyson, S. Jarvis, and G. Nudd. The Globus monitoring and discovery service (MDS-2): a performance analysis. *In Proc. 19th Annual UK Performance Engineering Workshop (UKPEW'2003), University of Warwick, UK*, pages 103–116, July 9-10, 2003.

[70] H. Lim Choi Keung, J. Dyson, S. Jarvis, and G. Nudd. Performance modelling of a self-adaptive and self-optimising resource monitoring system for dynamic Grid environments. *In Proc. UK e-Science All Hands Conference: e-Science Broadening the Horizon (AHM'2004), Nottingham, UK*, August 31 - September 3, 2004.

[71] H. Lim Choi Keung, J. Dyson, S. Jarvis, and G. Nudd. Self-adaptive and self-optimising resource monitoring for dynamic Grid environments. *In Proc. 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS'2004), in conjunction with the 15th International Conference on Database and Expert Systems Applications (DEXA'2004), Zaragoza, Spain*, pages 689–693, August 30 - September 4, 2004.

[72] H. Lim Choi Keung, L. Wang, D. Spooner, S. Jarvis, W. Jie, and G. Nudd. Grid resource management information services for scientific computing. *International Conference on Scientific & Engineering Computation (IC-SEC) 2002, Singapore*, pages 746–750, December 3-5, 2002.

[73] C. Liu and I. Foster. A constraint language approach to Grid resource selection. *Technical Report TR-2003-07, Department of Computer Science, University of Chicago*, March 31, 2003.

[74] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for Grid applications. *In Proc. 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC-11), Edinburgh, Scotland*, pages 63–72, July 23-26, 2002. ISBN 0-7695-1686-6, IEEE Computer Society.

[75] G. Marsaglia, A. Zaman, and W. Tsang. Toward a universal random number generator. *Letters in Statistics and Probability*, 9(1):35–39, January 1990.

[76] M. Massie, B. Chun, and D. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, June 2004.

[77] MDS 2.2: Creating a Hierarchical GIIS. `http://www.globus.org/toolkit/docs/2.4/mds/hierarchical_GIIS.pdf`, last accessed on September 3, 2005.

[78] MDS 2.2 GRIS Specification Document: Creating New Information Providers. `http://www.globus.org/toolkit/docs/2.4/mds/creating_new_providers.pdf`, last accessed on July 19, 2005., 2003.

[79] MDS 2.4 Configuration Files. `http://www.globus.org/toolkit/docs/2.4/mds/Config.html`, last accessed on July 19, 2005.

[80] MDS 2.4 Core GRIS Providers. `http://www.globus.org/toolkit/docs/2.4/mds/DefaultGRISProviders.html`, last accessed on July 19, 2005.

[81] MDS 2.4 Schemas. `http://www.globus.org/toolkit/docs/2.4/mds/Schema.html`, last accessed on August 28, 2005.

[82] MonALISA: MONitoring Agents using a Large Integrated Services Architecture. `http://monalisa.cacr.caltech.edu/`, last accessed on November 10, 2005.

[83] Namespaces in XML, World Wide Web Consortium 14-January-1999. `http://www.w3.org/TR/1999/REC-xml-names-19990114/#dt-qname`, last accessed on November 28, 2005.

[84] National Grid Service. `http://www.grid-support.ac.uk/`, last accessed on August 20 2006.

[85] H. Newman, I. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. MonALISA: A distributed monitoring service architecture. *Computing in High Energy and Nuclear Physics (CHEP'03), La Jolla, CA, USA*, March 2003.

[86] OASIS: Advancing e-business standards since 1993. `http://www.oasis-open.org/`, last accessed on November 12, 2005.

[87] OASIS UDDI: Advancing web services discovery standard. `http://www.uddi.org/`, last accessed on November 10, 2005.

[88] OpenLDAP: Community-developed LDAP software. `http://www.openldap.org/`, last accessed on November 16, 2005.

[89] OpenLDAP Schema Specification. `http://www.openldap.org/doc/admin/schema.html`, last accessed on July 21, 2005.

[90] OpenLDAP Software 2.3 Administrator's Guide. `http://www.openldap.org/devel/admin/guide.html`, last accessed on August 24, 2005.

[91] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *In Proc. 1st ACM Workshop on Hot Topics in Networks (HotNets-I), Princeton, New Jersey*, October 2002.

[92] PostgreSQL. `http://www.postgresql.org/`, last accessed on November 16, 2005.

[93] PostgreSQL JDBC Driver. `http://jdbc.postgresql.org/`, last accessed on November 16, 2005.

[94] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. *In Proc. 7th IEEE International Symposium on High-Performance Distributed Computing (HPDC-7), Chicago, Illinois*, pages 140–146, July 28-31, 1998. IEEE Computer Society.

[95] Realizing the information future: The internet and beyond. *National Academy Press, Washington, DC*, 1994.

[96] RFC 1832 - XDR: External Data Representation Standard. `http://www.faqs.org/rfcs/rfc1832.html`, last accessed on November 10, 2005.

[97] RRDTool Time-Series Data Archiving System and Graphing. `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/`, last accessed on November 10, 2005.

[98] F. Sacerdoti, M. Katz, M. Massie, and D. Culler. Wide area cluster monitoring with Ganglia. *In Proc of IEEE International Conference on Cluster Computing (Cluster'03), Hong Kong*, December 1-4, 2003.

[99] J. Schopf, I. Raicu, L. Pearlman, N. Miller, C. Kesselman, I. Foster, and M. d'Arcy. Monitoring and Discovery in a Web Services Framework: Functionality and Performance of Globus Toolkit MDS4. *Submitted to the 15th IEEE International Symposium on High-Performance Distributed Computing (HPDC-15), Paris, France*, January 2006. # MCS Preprint #ANL/MCS-P1315-0106.

[100] Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/soap/`, last accessed on November 9, 2005.

[101] W. Smith, A. Waheel, D. Meyers, and J. Yan. An evaluation of alternative designs for a Grid information service. *In Proc. 9th IEEE International Symposium on High-Performance Distributed Computing (HPDC-9)*, pages 185–192, August 1-4, 2000. IEEE Press.

[102] Specification: Web Services Inspection Language (WS-Inspection) 1.0. `http://www-128.ibm.com/developerworks/library/specification/ws-wsilspec/`, last accessed on November 10, 2005.

[103] Specification: Web services security (WS-Security), Version 1.0, April 5, 2002. `http://www-128.ibm.com/developerworks/library/ws-secure/`, last accessed on October 10 2005.

[104] D. Spooner, J. Cao, J. Turner, H. Lim Choi Keung, S. Jarvis, and G. Nudd. Localised workload management using performance prediction and QoS contracts. *In Proc. 18th Annual UK Performance Engineering Workshop (UKPEW'2002), University of Glasgow, UK*, pages 69–80, July 10-11, 2002.

[105] Storage Networking Industry Association. `http://www.snia.org`, last accessed on November 12, 2005.

[106] M. Swany and R. Wolski. Representing dynamic performance information in Grid environments with the Network Weather Service. *In Proc. 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'02), Berlin, Germany*, pages 48–56, May 21-24, 2002. ISBN 0-7695-1582-7, IEEE Computer Society.

[107] The DataGrid project. `http://eu-datagrid.web.cern.ch/eu-datagrid/`, last accessed on November 3, 2005.

[108] The EU DataTAG Project, Research and Technological Development for a Data TransAtlantic Grid. `http://datatag.web.cern.ch/datatag/`, last accessed on September 1, 2005.

[109] The Globus Alliance. `http://www.globus.org`, last accessed on September 5, 2005.

[110] The LDAP Data Interchange Format (LDIF) Technical Specification, RFC 2849, June 2000. `ftp://ftp.isi.edu/in-notes/rfc2849.txt`, last accessed on July 20, 2005.

[111] The MySQL Database. `http://mysql.com`, last accessed on November 10, 2005.

[112] The Network Weather Service. `http://nws.cs.ucsb.edu/`, last accessed on November 4, 2005.

[113] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A Grid monitoring architecture. *Informational Document, GGF Performance Working Group*, January 16, 2002.

[114] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. *In Proc. 7th IEEE High-Performance Distributed Computing Symposium (HPDC-7), Chicago, Illinois*, pages 260–267, July 28-31, 1998. IEEE Computer Society.

[115] UDDI Specification V3. `http://www.uddi.org/specification.html`, last accessed on November 10, 2005.

[116] UK eScience Programme, Research Councils UK. `http://www.research-councils.ac.uk/escience/`, last accessed on September 5, 2005.

[117] S. Vadhiyar and J. Dongarra. A performance oriented migration framework for the Grid. *In Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan*, pages 130–137, May 12-15 2003. ISBN 0-7695-1919-9, IEEE Computer Society.

[118] G. von Laszewski and I. Foster. Usage of LDAP in Globus. `http://www.cs.ucy.ac.cy/crossgrid/cygriddl/ldap_in_globus.pdf`, last accessed on September 5, 2005.

[119] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency Comput. Pract. Exp.*, 13(8-9):643–662, 2001.

[120] Web services architecture, W3C Working Group Note 11 February 2004. `http://www.w3.org/TR/ws-arch/`, last accessed on November 9, 2005.

[121] Web services description language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, last accessed on November 9, 2005.

[122] The White Rose Grid e-Science Centre. `http://www.wrgrid.org.uk/`, last accessed on August 20 2006.

[123] R. Wolski. Forecasting network performance to support dynamic scheduling using the Network Weather Service. *In Proc. 6th IEEE International Symposium on High-Performance Distributed Computing (HPDC-6), Portland, Oregon*, pages 316–325, August 5-8, 1997. ISBN 0-8186-8117-9, IEEE Computer Society Press.

[124] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[125] R. Wolski, J. Plank, J. Brevik, and T. Bryan. G-commerce: Market formulations controlling resource allocation in the computational Grid. *In Proc. of 15th International Parallel and Distributed Processing Symposium (IPDPS'2001), San Francisco, CA*, April 23-27 2001. ISBN 0-7695-0990-8, IEEE Computer Society.

[126] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.

[127] Work Package 3: Information and Monitoring Services, EU DataGrid Project. `http://hepunx.rl.ac.uk/edg/wp3/`, last accessed on November 7, 2005.

[128] Writing a Custom Information Provider in MDS 2.x from Start to Finish. `http://www.thecodefactory.org/mds`, last accessed on August 28, 2005.

[129] X. Zhang, J. Freschl, and J. M. Schopf. A performance study of monitoring and information services for distributed systems. *In Proc. 12th IEEE International Symposium on High-Performance Distributed Computing (HPDC-12)*, pages 270–281, June 22-24, 2003. IEEE Press.